

Optimalizace datové vrstvy aplikací pro PostgreSQL

Optimization of a Data Layer of an Application Using PostgreSQL

Zadání diplomové práce

Student:

Bc. David Krutký

Studijní program:

N2647 Informační a komunikační technologie

Studijní obor:

2612T025 Informatika a výpočetní technika

Téma:

Optimalizace datové vrstvy aplikací pro PostgreSQL
Optimization of a Data Layer of an Application Using PostgreSQL

Zásady pro vypracování:

Cílem práce je vytvořit rámec pro datovou vrstvu informačních systémů využívajících SŘBD PostgreSQL. Rámec bude usnadňovat optimalizaci jednotlivých operací.

Funkce, které budou v rámci implementovány:

1. Simulace editovatelných pohledů v PostgreSQL.
2. Automatická denormalizace původního návrhu.

Práce bude mít následující části:

1. Řešení různých řešení, které se používají pro realizaci datové vrstvy v IS.
2. Návrh a implementace funkcí vlastního rámce.
3. Porovnání výsledného rámce s existujícími rámci.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013



doc. Dr. Ing. Eduard Sojka
vedoucí katedry



prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 7. května 2013

..........

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2013

..........

Rád bych na tomto místě poděkoval všem, kteří mi k vytvoření práce dopomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Tato diplomová práce se zaměřuje na problém, který představuje komunikace aplikačního softwaru s relačním databázovým systémem. Popisuje datovou vrstvu aplikačního softwaru, její funkce a vlastnosti. Rozebírá objektově-relační mapování a návrhové vzory využívané při použití tohoto mapování. Dále pak seznamuje čtenáře s existujícími technologiemi pracujícími na datové vrstvě vytvořenými v jazyce Python a zaměřuje se na optimalizační techniky používané pro zefektivnění komunikace s relační databází. Výstupem této práce je knihovna optimalizovaná pro komunikaci s relačním databázovým systémem PostgreSQL. V práci je popsán návrh této knihovny a optimalizačních technik. V závěru je pak vytvořená knihovna srovnána s existujícími technologiemi pomocí rychlostních testů.

Klíčová slova: Datová vrstva, Objektově-relační mapování, Optimalizace, SŘBD, PostgreSQL, Python, Django ORM, web2py DBAL, SQLAlchemy

Abstract

This master's thesis focuses on difficulties, which appear when application software is used with relational database system. It contains a description of the application data access layer and its characteristics, object-relational mapping and design patterns. Existing technologies used for data access created with Python programming language are described in this thesis. Furthermore, techniques optimizing the communication with relational database are discussed. The goal of this thesis was to create a new software for the application data layer, that would focus on optimization of DBMS PostgreSQL usage. The thesis goes through the design of this software and its techniques. Finally, the software is compared to the existing technologies, using benchmarks.

Keywords: Data access layer, Object-relational mapping, Optimization, DBMS, PostgreSQL, Python, Django ORM, web2py DBAL, SQLAlchemy

Seznam použitých zkratek a symbolů

SŘBD	–	Systém řízení báze dat
DBMS	–	Database Management System
DAL	–	Data-Access Layer
DBAL	–	Database-Abstraction Layer
SQL	–	Structured Query Language
ORM	–	Object-Relational Mapping/Mapper
ERD	–	Entity-Relationship Diagram
MVC	–	Model-View-Controller
API	–	Application Programming Interface
IDE	–	Integrated Development Environment
XML	–	Extensible Markup Language
HTML	–	HyperText Markup Language
CSV	–	Comma Separated Values
IS	–	Informační systém
CMS	–	Content Management System
SSOT	–	Single Source Of Truth
JSON	–	JavaScript Object Notation
POPO	–	Plain Old Python Object

Obsah

1	Úvod	6
2	Vícevrstvá architektura	7
2.1	Třívrstvá architektura	7
2.2	Datová vrstva aplikačního softwaru	8
3	Objektově relační mapování	10
3.1	Rozhraní datové vrstvy	10
3.2	Mapování vztahů	13
3.3	Mapování dědičnosti	14
4	Existující technologie pracující na datové vrstvě	17
4.1	Django ORM	17
4.2	web2py DBAL	22
4.3	SQLAlchemy	25
4.4	Praktické srovnání existujících technologií pracujících na datové vrstvě . .	31
5	Specifikace požadavků pro tvorbu nové technologie	35
5.1	Objektově-relační mapování	36
5.2	Podpora různých SŘBD	36
5.3	Funkce pro zlepšení výkonu při čtení z databáze	37
6	Návrh nové technologie pracující na datové vrstvě	41
6.1	Klíčové vlastnosti	41
6.2	Architektura	42
6.3	Funkce pro zrychlení čtení dat	44
6.4	Návrh syntaxe	47
6.5	Funkce, které budou v budoucnu implementovány	48
7	Porovnání nové technologie s ostatními rámci	51
7.1	Podmínky testování	51
7.2	Popis testů	53
7.3	Vytváření testů	54
7.4	Výsledky testů	55
8	Závěr	62
9	Reference	63
	Přílohy	65
A	Výpisy mapování rámců ze sekce 4.4	66
B	Výpisy SQL vygenerované rámci ze sekce 4.4	70

C	Výstup automatické denormalizace knihovny pypg	73
D	Obsah přiloženého CD	74

Seznam tabulek

1	Ukázka multiplicity dat při spojení tabulek	39
2	Výsledky testování vlivu objemu přenášených dat na celkový výkon . . .	39
3	Počet záznamů v tabulkách testovací databáze	52
4	Tabulka výsledků testu 1 - detail produktu	55
5	Tabulka výsledků testu 2 - detail zákazníka	56
6	Tabulka výsledků testu 3 - detail objednávky	57
7	Tabulka výsledků testu 4 - produkty v kategorii	57
8	Tabulka výsledků testu 5 - objednávky zákazníka	60
9	Tabulka výsledků testu 6 - vytvoření nového zákazníka a objednávky . . .	61

Seznam obrázků

1	Rozdělení existujících rámců a knihoven pracujících na datové vrstvě . . .	9
2	Třída v návrhovém vzoru Table Data Gateway [1, str. 148]	11
3	Ukázka použití návrhového vzoru Row Data Gateway [1, str. 154]	12
4	Ukázka použití návrhového vzoru Data Mapper [1, str. 164]	14
5	Ukázka návrhových vzorů pro mapování dědičnosti [1, str. 78-79]	15
6	Architektura rámce SQLAlchemy [28, str. 3]	27
7	ERD diagram jednoduché aplikace pro vytváření blogů	31
8	Diagram tříd knihovny pypg	43
9	Diagram aktivit zobrazující vytváření materializovaného pohledu	45
10	Diagram aktivit zobrazující procesy automatické denormalizace	46
11	ERD diagram testovací databáze	52
12	Graf výsledků testu 1 - detail produktu	55
13	Graf výsledků testu 2 - detail zákazníka	56
14	Graf výsledků testu 3 - detail objednávky	58
15	Graf výsledků testu 4 - produkty v kategorii	59
16	Graf výsledků testu 5 - objednávky zákazníka	60
17	Graf výsledků testu 6 - vytvoření nového zákazníka a objednávky	61

Seznam výpisů zdrojového kódu

1	N+1 problém a jeho řešení v Django ORM	20
2	Příklad dotazu, který nelze provést pomocí Django ORM API	22
3	Ukázka mapování v Django ORM	32
4	Ukázka mapování ve web2py DBAL	32
5	Ukázka mapování v SQLAlchemy	32
6	Ukázka práce s daty v Django ORM	33
7	Ukázka práce s daty ve web2py DBAL	33
8	Ukázka práce s daty v SQLAlchemy	34
9	Získávání dat pomocí samostatných dotazů	40
10	Návrh implementace samostatných dotazů v pypg	47
11	Dotazy vygenerované z výpisu 10 knihovnou pypg	47
12	Návrh syntaxe knihovny pypg	48
13	Minimalizace přenášených dat z tabulky 1 pomocí formátu JSON	50
14	Schéma jednoduché aplikace pro vytváření blogů v Django ORM	66
15	Schéma jednoduché aplikace pro vytváření blogů ve web2py DBAL	67
16	Schéma jednoduché aplikace pro vytváření blogů v SQLAlchemy	68
17	Reprezentace modelů z výpisu 14 v PostgreSQL vygenerovaná rámcem Django ORM	70
18	Reprezentace schéma z výpisu 15 v PostgreSQL vygenerovaná rámcem web2py DBAL	71
19	Reprezentace schéma z výpisu 16 v PostgreSQL vygenerovaná rámcem SQLAlchemy	72
20	Výstup automatické denormalizace po testech popsáných v kapitole 7 . . .	73

1 Úvod

Aplikační softwary dnes ve velké míře používají pro ukládání dat systémy řízení báze dat (SŘBD). Velmi rozšířeným typem databázových systémů jsou relační databáze. Při použití relačních databází ovšem nastává mnoho problémů, pokud je aplikační software vyvinut pomocí objektově orientovaného jazyka. Relační databáze totiž pro práci s daty používají dotazovací jazyky, které nejsou velmi kompatibilní s objektově orientovanými jazyky. Tyto dvě technologie se mezi sebou liší hlavně datovými typy, strukturami dat a řešením relací mezi entitami.

Relační databázi tedy nelze v aplikačním softwaru použít přímo. Proto dnes většina systémů obsahuje mezivrstvu, která slouží jako prostředník mezi těmito technologiemi. Tato vrstva se nazývá datová vrstva. Datová vrstva řeší tuto nekompatibilitu, abstrahuje relační databázový systém a poskytuje objektově orientované aplikační rozhraní. Jelikož je relační databáze většinou nejzatěžovanější částí v celém systému, je také důležité, aby byla datová vrstva výkonná a komunikovala s databází efektivně.

Tato diplomová práce se zaměřuje na technologie vytvořené pro realizaci datové vrstvy. Popisuje všechny vlastnosti této vrstvy a porovnává existující technologie pracující na této vrstvě. Dále se pak soustředí na výkon a optimalizaci komunikace s relační databází. Praktická část práce sestává z implementace knihovny pro datovou vrstvu orientující se na SŘBD PostgreSQL a zaměřující se na optimalizaci komunikace s touto SŘBD.

V kapitole 2 je popsáno rozdělení architektury aplikačního softwaru do vrstev. Je zde popsána datová vrstva a jsou zde vysvětleny základní pojmy spojené s přístupem k relačním databázím pomocí objektově orientovaných jazyků.

Kapitola 3 popisuje velmi používané řešení pro datovou vrstvu, objektově-relační mapování. Jsou zde popsány různé přístupy mapování a návrhové vzory, kterými se řídí rámce implementující objektově-relační mapování.

Kapitola 4 obsahuje srovnání a popis existujících řešení pracujících na datové vrstvě vytvořených v jazyce Python. Jsou zde dopodrobna rozebrány rámce Django ORM, web2py DBAL a SQLAlchemy. U každého rámce jsou napsány jeho nevýhody.

V kapitole 5 jsou popsány důvody vytvoření nové knihovny, požadavky na její implementaci a funkce, které by vytvořená knihovna měla obsahovat. Je zde shrnuto také, proč nevystačovaly existující rámce.

Návrh knihovny popisuje kapitola 6. Tato kapitola rozebírá návrh aplikačního rozhraní této knihovny a klíčové funkce, které bude knihovna obsahovat. Dále je zde analyzována architektura knihovny a popis návrhu řešení pro optimalizaci komunikace se SŘBD PostgreSQL.

Kapitola 7 pak srovnává rámce z kapitoly 4 a vytvořenou knihovnu pomocí rychlostních testů.

Pro dokonalé porozumění textu je vhodná alespoň minimální znalost relačních databází a programovacího jazyka Python.

2 Vícevrstvá architektura

Použití vrstev je jedna z nejčastějších technik využívaných pro zjednodušení komplexnosti složitých systémů [1]. Vrstvy ve vrstvené architektuře popisují logické seskupení funkcí a komponent v aplikaci. Shlukují komponenty podle typu funkcí a ulehčují tak jejich znovupoužitelnost. Vrstvy přitom nemusí být ani fyzicky oddělené, pouze vymezují a rozdělují aplikaci do logických celků. Každá z vrstev této architektury komunikuje pouze se svými sousedními vrstvami a nestará se o další části systému. Vrstva na nižší úrovni poskytuje rozhraní, které je známé vrstvě o úroveň výše. Tato architektura tak umožňuje jednoduše nahradit nebo změnit jednotlivé vrstvy, aniž by to ovlivnilo ostatní vrstvy.

Při vytváření aplikace založené na této architektuře je tak možno postupovat ve vývoji jednotlivých vrstev souběžně. Tohoto je dosaženo pomocí předem určeného rozhraní. Mezi další výhody patří také jednodušší údržba a testování, kde lze využít i tzv. mock-up testů.

Nevýhodou vrstev je, že při vysokém počtu se můžou velmi negativně projevat na výkonu celého systému. Nejvýraznějším vlivem je potřeba rozhraní pro převod dat, jelikož každá z vrstev má vlastní logiku a interpretuje data vlastním způsobem. Dalším problémem jsou změny, které ovlivňují všechny vrstvy o úroveň výše tzv. kaskádové změny [5]. V informačních systémech to může být například přidání databázového sloupce.

Vrstvenou architekturu lze aplikovat nezávisle na typu aplikace [2]. Příkladem použití této architektury je referenční model ISO/OSI¹.

2.1 Třívrstvá architektura

Jedním z nejrozšířenějších typů vrstvené architektury v aplikacích je třívrstvá architektura. Mezi vrstvy třívrstvé architektury patří vrstvy prezentační, aplikační a datová.

Úkolem prezentační vrstvy je zobrazovat uživateli data ve srozumitelné a přehledné formě, reagovat na jeho požadavky a propagovat provedené změny. Prezentační vrstva může mít podobu uživatelského rozhraní v aplikaci, webové stránky nebo i jednoduchého rozhraní příkazové řádky.

Aplikační vrstva je srdcem aplikace. V této vrstvě se vyskytuje veškerá aplikační logika, pracuje se zde s daty, která se validují, provádějí se na nich výpočty a připravují se k uložení nebo k zobrazení. Spadají zde funkce jako autentifikace, autorizace, logování a kešování.

Datová vrstva zajišťuje manipulaci se zdrojem dat nebo datovým úložištěm a s daty, která tato úložiště poskytují. Abstrahuje a centralizuje přístup k tomuto úložišti dat. Důležitým úkolem datové vrstvy je také bezpečnost dat a jejich ochrana před případnými pokusy, které se snaží k těmto datům získat přístup nebo je poškodit. Datová vrstva by se také měla snažit o co nejefektivnější výkonnost a škálovatelnost, jelikož často bývá nejvíce zatěžovaným místem v aplikaci.

¹[http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC.7498-1_1994\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC.7498-1_1994(E).zip)

2.2 Datová vrstva aplikačního softwaru

Velice často je třeba, aby data zpracovávaná aplikací byla perzistentní. Tímto pojmem se rozumí trvalé zachování dat po zastavení provozu aplikace nebo i po výpadku systému. Jako nástroje poskytující tuto funkci jsou dnes hojně rozšířené relační databáze.

Princip datové vrstvy při použití relačních databází zůstává stejný. Mezi její funkce patří:

- správa spojení,
- řízení paralelního zpracování a transakce,
- synchronizace konkurenčních dat v aplikaci a v databázi,
- validace dat,
- formátování dat,
- dotazování a získávání dat z databáze,
- persistence dat,
- správa chyb a chybových hlášení,
- zajištění bezpečnosti dat,
- dávkování dotazů,
- stabilita a efektivní výkon.

Datová vrstva musí také řešit problém, který představuje nekompatibilita objektových jazyků použitých pro aplikační logiku a dotazovacích jazyků využívaných pro získávání dat z relačních databází. Tento problém se označuje pod pojmem *Impedance mismatch*. Do tohoto problému spadá například převod syntaxe, převod datových typů, efektivní interpretace relací a struktur a jejich mapování mezi dvěma nekompatibilními jazyky.

Impedance mismatch se začal více projevovat až při použití objektově orientovaných jazyků. Jazyky používané dříve měly specializované datové struktury optimalizované pro zacházení s relačními daty, kdežto objektově orientovaným jazykům tyto struktury schází [4].

Pro zjednodušení a odstínění od těchto problémů byly vytvořeny rámce a knihovny, které se starají o tyto monotónní činnosti. Rozdíl mezi knihovnou a rámcem definuje Martin Fowler pomocí *Inversion of control*. Knihovna poskytuje funkce, které je možné v aplikační logice využívat. Je tedy řízena aplikační logikou. Naopak při použití rámce je třeba umístit aplikační kód na předem určená místa například pomocí dědičnosti. Tento kód je pak volán a řízen rámcem [3].

Datová vrstva se v angličtině označuje pod názvem *Data-Access Layer* (DAL). Tímto pojmem se také někdy nazývají knihovny a rámce, které zastávají funkci této vrstvy. Často je tento název zaměňován s *Database-Abstraction Layer* (DBAL). DBAL je typ DAL, který

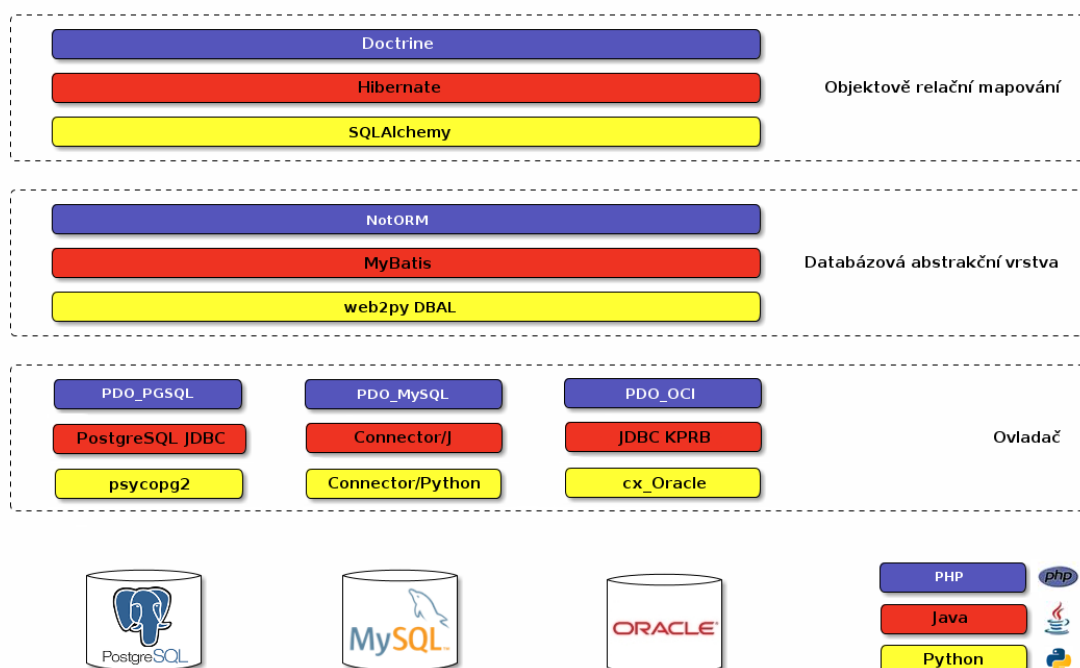
pomocí svých funkcí abstrahuje rozdíly mezi detaily a specifikacemi podporovaných SŘBD. Umožňuje tak jednodušeji změnit SŘBD bez velkých úprav aplikačního kódu.

Samotná datová vrstva je velmi často rozdělena na další podvrstvy. Mezi nejprimitivnější DAL patří knihovny, které označujeme za konektory nebo ovladače. Řeší pouze část problému Impedance mismatch, poskytují základní funkce a zaměřují se pouze na jeden SŘBD. Zpravidla umožňují zadat pouze syrové SQL ve formě řetězce. Tyto konektory jsou většinou vytvářeny výrobcem SŘBD a řídí podle specifikovaných standardů. Mnoho programovacích jazyků má určeno tyto standardy proto, aby sjednotily rozhraní a práci s těmito knihovnami, což umožňuje lehčí přechod mezi SŘBD a jednodušší vytváření DBAL. Příkladem standardu pro jazyk Python je specifikace PEP 249².

Konektory pak využívají již zmíněné DBAL a složitější rámce a knihovny snažící se o větší míru abstrakce. Tyto tak tvoří další vrstvu.

Nad touto vrstvou je dnes velmi často postavena ještě další funkčnost mající za cíl problém Impedance mismatch ještě více ulehčit a abstrahovat. Touto funkčností je objektově relační mapování, v angličtině *Object-Relational Mapping* (ORM). ORM lze považovat za nejvyšší míru abstrakce ze všech typů DAL.

Rozdělení existujících rámců a knihoven do těchto tří vrstev znázorňuje obrázek 1.



Obrázek 1: Rozdělení existujících rámců a knihoven pracujících na datové vrstvě

²<http://www.python.org/dev/peps/pep-0249/>

3 Objektově relační mapování

Objektově relační (O/R) mapování, jak už název napovídá, mapuje entity z relačního modelu na doménové objekty v aplikační logice nejčastěji pomocí objektů nebo tříd.

Pod pojmem doména rozumíme u aplikace oblast z reálného světa, pro kterou se tato aplikace snaží vyřešit nebo zjednodušit specifický problém. Doména vymezuje rozsah působnosti aplikace a určuje doménové objekty. Tyto objekty se v objektově orientovaném jazyce modelují nejčastěji pomocí tříd. Například, pokud je doména nákup a prodej zboží, za doménové objekty lze považovat kupujícího a objednávku.

Martin Fowler definuje ve své publikaci několik návrhových vzorů, které popisují modelování doménových objektů a přístup k doménové logice. Mezi tyto vzory patří *Transaction Script*, *Domain Model* a *Table Module*. Popis těchto vzorů je nad rámec této diplomové práce. Tyto návrhové vzory se však berou v potaz při návrhu objektově relačního mapování. Více informací o těchto vzorech lze najít v knize Martina Fowlera [1].

Podobně jako u DAL se i pod zkratkou ORM někdy označují rámce implementující O/R mapování. Tyto rámce kromě O/R mapování obsahují funkce potřebné pro komunikaci s relační databází popsané výše v sekci 2.2. Někdy i využívají existujících DAL, které tyto funkce obsahují a přidávají k nim O/R mapování. Většinou obsahují také DBAL.

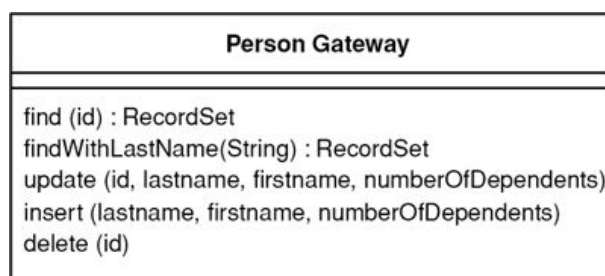
3.1 Rozhraní datové vrstvy

Každé ORM definuje způsob, kterým datová vrstva komunikuje s aplikační vrstvou. Tímto rozhraním ORM také abstrahuje přístup k databázi a z části určuje i architekturu doménové logiky. Martin Fowler pojmenoval několik základních typů a návrhových vzorů pro toto rozhraní jako *Table Data Gateway*, *Row Data Gateway*, *Active Record* a *Data Mapper* [1]. Ve všech případech jsou v objektově orientovaném jazyce jako rozhraní využity třídy, pomocí kterých ORM abstrahuje tabulky, pohledy, dynamické dotazy nebo dotazy zapouzdřené v uložených procedurách. Vzory se mezi sebou liší typem abstrakce a způsobem, jakým jsou tyto třídy využity.

3.1.1 Table Data Gateway

Table Data Gateway využívá třídy jako brány k tabulkám. Tabulka nebo podobný databázový objekt je namapována pomocí třídy. Třída obsahuje metody pouze pro ukládání, mazání a vyhledávání v datech tabulky. Jedna instance této třídy přitom poskytuje rozhraní ke všem záznamům v tabulce a ke konkrétnímu řádku tabulky se lze dostat jen předáním klíče, jedinečného identifikátoru. Tento návrhový vzor je ve většině případů bezstavový.

Každá metoda třídy abstrahující tabulku skrývá za svým rozhraním SQL. Nejčastěji komunikuje přímo s databázovým ovladačem a předává mu vytvořené SQL. Data jsou pak vrácena ve formě, v jaké je vrací ovladač, v objektu typu *Record Set* [1]. Record Set je generická reprezentace relačních dat v paměti, většinou podobná poli nebo seznamu. Metody pro vyhledávání v datech pak vracejí vždy tento objekt, i pokud se jedná pouze o jediný záznam. Vracená data navíc neobsahují žádnou referenci k tomuto rozhraní. Toto



Obrázek 2: Třída v návrhovém vzoru Table Data Gateway [1, str. 148]

může být problém, a proto se vrácená data někdy obalují dalším objektem. Tento objekt se označuje jako *Data Transfer Object* [1].

Výhodou při použití tohoto vzoru je, že jeho rozhraní lze využít jak pro manipulaci relačních dat, tak pro uložené procedury. Také shlukuje SQL na jedno místo, což zjednodušuje optimalizaci. Table Data Gateway lze lehce využít jako rozhraní datové vrstvy při použití vzorů Table Module nebo Transaction Stript v aplikační logice.

Příkladem tohoto návrhového vzoru je obrázek 2. Na obrázku je třída `PersonGateway`, která slouží jako brána k databázové tabulce `Person`.

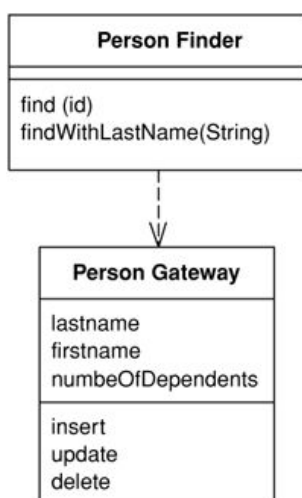
3.1.2 Row Data Gateway

Row Data Gateway abstrahuje tabulku pomocí třídy, její záznamy pomocí instancí této třídy a její sloupce pomocí atributů. Třídy neobsahují žádnou aplikační logiku a slouží pouze jako brána k datům. Jedna instance odpovídá jednomu záznamu. Lze se tak jednoduše dostat ke konkrétnímu řádku. Instance má navíc povědomí o celém rozhraní, není tedy třeba používat Data Transfer Object.

Instance obsahuje metody pro manipulaci s daty řádku, jejich mazání a ukládání. Datové typy atributů třídy jsou implicitně převáděny na datové typy konkrétních sloupců tabulky. Na základě tříd a jejich atributů, lze i jednoduše generovat pomocí nástrojů ORM relační schéma v databázi.

Ke každé třídě mapující tabulku je vytvořena další třída, která slouží k vyhledávání záznamů v této tabulce. Tato třída se v pozadí chová podobně jako třída v návrhovém vzoru Table Data Gateway, a je tak možné využívat i některých výhod tohoto vzoru. Namísto objektu Record Set však třída pro vyhledávání vrací instanci třídy abstrahující tabulku. Pro získání záznamu je tedy nejdříve nutné využít vyhledávací třídu.

Příklad návrhového vzoru Row Data Gateway zobrazuje obrázek 3. Na obrázku je třída `PersonFinder`, která vyhledává data v databázové tabulce `Person` a vrací instanci třídy `PersonGateway`. `PersonGateway` pak slouží jako rozhraní pro konkrétní řádek tabulky `Person`.



Obrázek 3: Ukázka použití návrhového vzoru Row Data Gateway [1, str. 154]

3.1.3 Active Record

Active Record je návrhový vzor podobný Row Data Gateway. Stejně jako Row Data Gateway mapuje struktury v datové vrstvě velmi úzce se strukturou databáze. Třídy tak korespondují s tabulkami v databázi a jejich instance a atributy s řádky a sloupci tabulek. Oproti Row Data Gateway však třídy obsahují navíc i aplikační logiku a většinou jsou tak považovány i za doménové objekty.

„Třída ve vzoru Active Record typicky obsahuje metody, které mají za úkol následující:

- Vytvořit instanci třídy z dat řádku vráceného na SQL dotaz
- Vytvořit instanci třídy, která bude později použita pro vložení nového záznamu do tabulky
- Statické metody, které obalují často používané SQL dotazy pro vyhledávání dat a které vracejí instance třídy
- Aktualizovat databázi daty instance
- Metody pro získání a nastavování hodnot atributů
- Implementace části aplikační logiky“ [1, str. 160]

Tento návrhový vzor je intuitivním přístupem, pokud při návrhu databázového schématu korespondují tabulky s doménovými objekty. Proto a pro svou jednoduchost je dnes velmi rozšířeným a často používaným vzorem při návrhu ORM. Používá se hlavně pro jednoduché systémy, kterým umožňuje rychlou implementaci rozhraní datové vrstvy.

Mezi nevýhody návrhového vzoru Active Record patří mapování vztahů a dědičnosti, které zčásti rozbíjí jeho principy a přispívají tak ke špatné struktuře kódu a chybám, které z toho plynou. Podobný problém nastává při složitější aplikační logice, jelikož pak

třídy výrazně nabudou na objemu a zastávají nemalý počet funkcí. Toto vede ke špatně testovatelným objektům a nevyhnutelně k chybám. Active Record se nevyplatí používat, pokud se doménové objekty liší od struktur v databázi nebo pokud by v budoucnu mohlo k takovýmto změnám dojít.

3.1.4 Data Mapper

Data Mapper se využívá při složitější aplikační logice, při rozdílném objektovém a relačním schématu nebo i často v případech, kdy je potřeba pracovat s již existující databází. Doménové objekty v aplikační logice se při použití tohoto vzoru nemusí shodovat s databázovým schématem. Lze je vytvořit naprosto nezávisle. Je tak možné kdykoliv změnit databázové schéma nebo struktury v aplikační logice, z čehož vyplývá mnoho výhod.

Nevýhodou tohoto přístupu je přidání další vrstvy mezi databázi a aplikační vrstvu. Toto se může negativně projevovat na celkovém výkonu a složitosti systému. Proto se Data Mapper nedoporučuje používat pro jednoduché případy, které lze vyřešit například pomocí vzoru Active Record.

Data Mapper komunikuje s aplikační vrstvou pomocí mapovací třídy, která poskytuje rozhraní k získávání doménových objektů. Mapovací třída může abstrahovat jednu databázovou tabulku, její část nebo i více tabulek. Většinou je výhodnější pro jednu doménovou třídu vytvořit vždy jednu mapovací třídu.

Obrázek 4 zobrazuje jednoduchý příklad návrhového vzoru Data Mapper. Na obrázku je mapovací třída `PersonMapper`, která získává data z databáze a vrací je ve formě instancí doménové třídy `Person`. Třída `PersonMapper` poskytuje metody i pro ukládání a mazání těchto instancí. Implementace získávání dat je před třídou `Person` skrytá a databázové schéma může vypadat jakkoliv.

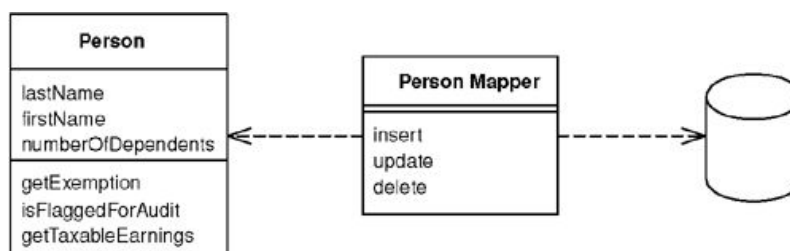
Data Mapper využívá pro správu doménových objektů strukturu pojmenovanou jako *Identity Map* [1]. Tato struktura se stará o synchronizaci doménových objektů a dat v paměti. Předchází provádění zbytečných dotazů do databáze, pokud daný objekt již v paměti existuje a zajišťuje, aby v takovém případě místo nového objektu byla vrácena reference na existující objekt.

Tento návrhový vzor má smysl používat jen tehdy, pokud se aplikační logika řídí vzorem Domain Model. Při použití jiných vzorů přináší Data Mapper mnoho nadbytečných komplikací a jeho použití se nevyplácí.

3.2 Mapování vztahů

V relační databázi se používají cizí klíče pro vyjádření relací, ale objekty v objektově orientovaných jazycích dosahují vazeb za pomoci referencí na vazební objekt. ORM tento nesoulad řeší mapováním vztahů.

Mapování vztahů a relací patří mezi základní funkce ORM. ORM řídící se podle návrhových vzorů Row Data Gateway, Active Record a Data Mapper tuto funkci potřebují, jelikož mapují instance tříd, které abstrahují databázové tabulky, na záznamy těchto tabulek.



Obrázek 4: Ukázka použití návrhového vzoru Data Mapper [1, str. 164]

ORM poskytuje mapování pro všechny druhy binárních relací.

Relace N:1 nebo 1:1 se mapuje nejjednodušeji. Řádek s cizím klíčem v relační databázi je namapován pomocí objektu, který obsahuje referenci na objekt mapující záznam odkazovaný cizím klíčem.

Relace 1:N je v relační databázi řešena opačným způsobem než v objektově orientovaných jazycích. Záznamy databázové tabulky Město obsahují cizí klíč na tabulku Stát, ale instance třídy Stát obsahují kolekci s referencemi na objekty třídy Město. ORM mapováním zajišťuje naplnění této kolekce většinou pomocí dodatečného dotazu do databáze, anebo spojením tabulek ihned při získávání objektu třídy Stát.

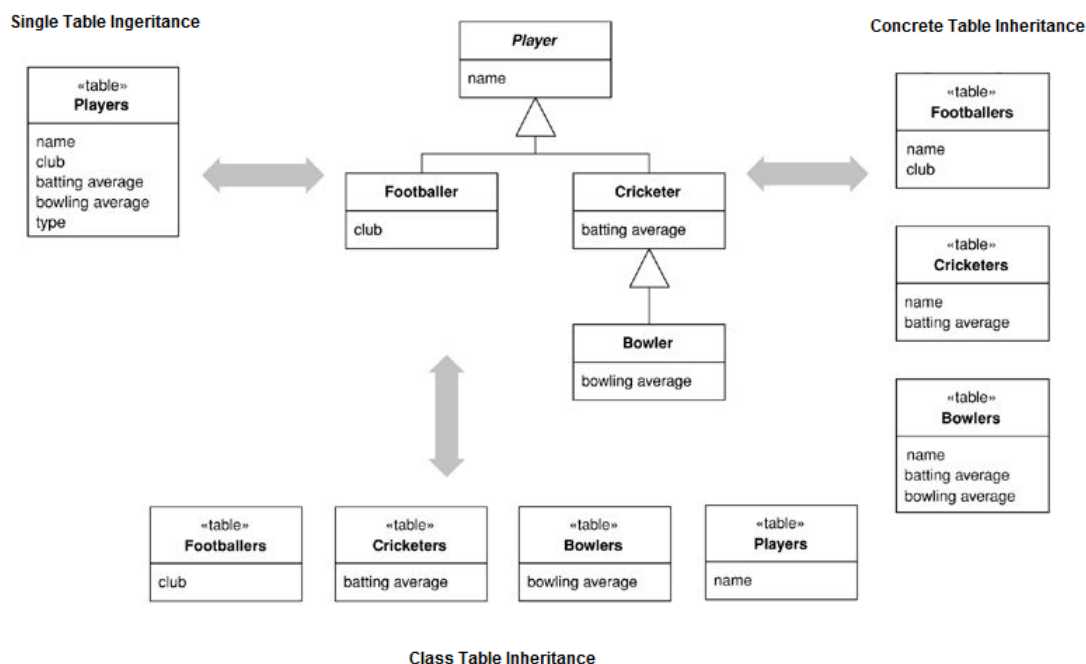
Relace N:M se v relační algebře řeší pomocí vazební tabulky, v objektově orientovaném jazyce to však potřeba není. Objekty mohou totiž jako v případě 1:N obsahovat kolekce s referencemi na vazební objekty. Úkolem ORM je tyto kolekce naplnit, což je zajištěno opět spojením tabulek nebo dodatečným dotazem. Komplikované je v tomto případě i zabezpečení integrity dat. ORM by mělo integritu dat zajišťovat, a to nejlépe za pomoci Identity Map.

Problémem, který musí ORM v souvislosti s mapováním vazeb řešit je nahrávání vazebních objektů do paměti. Nelze všechny vazby nahrávat najednou, protože hrozí, že by se do paměti načetlo velké množství dat, v nejhorším případě celá databáze. ORM proto většinou poskytuje funkci *Lazy Loading*. Lazy Loading zajišťuje, že při načtení objektu z databáze do paměti se data, která tvoří relaci, nenahrají ihned, ale až při přístupu k dané relaci.

Toto chování však také není vždy ideální, jelikož v určitých případech tak ORM může generovat mnoho přístupů do databáze, které by se daly jinak ušetřit. ORM by proto mělo sledovat, jak aplikační vrstva pracuje s objekty a podle toho nahrávat jejich vazby.

3.3 Mapování dědičnosti

Objektově relační mapování definuje principy, pomocí kterých je možné mapovat dědičnost objektově orientovaných jazyků v relačních databázích. V současné době většina databází dědičnost přímo nepodporuje a neexistuje standard, který by tuto funkčnost pro databáze specifikoval. ORM proto často obsahuje funkce, které umožňují dědičnost různými způsoby simulovat.



Obrázek 5: Ukázka návrhových vzorů pro mapování dědičnosti [1, str. 78-79]

Dědičnost je jedním z hlavních prvků objektově orientovaných jazyků a prostřednictvím ORM je možné ji využívat při definici tříd, kterými ORM abstrahuje databázové tabulky. Mapování dědičnosti je možné dosáhnout třemi způsoby, které Martin Fowler pojmenovává jako *Single Table Inheritance*, *Concrete Table Inheritance* a *Class Table Inheritance* [1].

Nevýhodou tohoto mapování je složitější struktura tabulek a z toho plynoucí zhoršení výkonu operací prováděných nad databází. Obrázek 5 zobrazuje příklady těchto tří návrhových vzorů.

3.3.1 Single Table Inheritance

Single Table Inheritance, někdy také mapování pomocí unie, používá jednu databázovou tabulku pro celou hierarchii tříd. Atributy této tabulky tvoří unii atributů ze všech tříd v hierarchii. Tabulka navíc obsahuje sloupec, který u každého záznamu určuje k jaké třídě v hierarchii náleží.

Výhodou tohoto přístupu je, že se při získávání dat nemusí používat spojení tabulek, jednoduchá implementace a vysoká tolerance vůči změnám struktury tříd.

Jelikož tabulka obsahuje unii všech atributů, spousta sloupců často zůstává prázdných a její velikost se může projevit negativně na výkonu. Nelze také využít integritních omezení pro atributy, které nejsou sdíleny všemi třídami v hierarchii.

3.3.2 Concrete Table Inheritance

Concrete Table Inheritance definuje pro každou konkrétní třídu jednu tabulku v databázi. Tabulka obsahuje všechny atributy třídy a navíc i všechny atributy, které třída získala pomocí dědění. Konkrétní třídou je myšlena každá třída, která dědí z abstraktní třídy, ale sama abstraktní třídou není. Většinou jsou to třídy v hierarchii na pozici listu.

Při získávání záznamů není třeba spojení tabulek, mezi nevýhody však patří velká citlivost na změny. Při změně struktury nadtřídy je třeba změnit všechny tabulky mapující třídy, které z této nadtřídy dědí. Navíc jsou zde primární klíče rozděleny do všech tabulek, což znesnadňuje jejich správu a znemožňuje vazby pomocí cizích klíčů.

Concrete Table Inheritance se někdy označuje také jako horizontální mapování.

3.3.3 Class Table Inheritance

Při použití vzoru Class Table Inheritance je vytvořena tabulka v databázi pro každou třídu. Tato tabulka však obsahuje atributy, které jsou definované pouze v dané třídě. Výhodou je velmi jednoduché vytvoření tohoto mapování. Tabulka neobsahuje žádné prázdné sloupce a primární klíče jsou uloženy na jednom místě. Nevýhodou je složitost dotazů a jejich pomalé vyhodnocování, jelikož dochází ke spojování tabulek.

Jiným názvem pro Class Table Inheritance bývá také Joined Table Inheritance nebo vertikální mapování.

4 Existující technologie pracující na datové vrstvě

V této kapitole budou popsány existující rámce a knihovny pracující na datové vrstvě vytvořené pomocí jazyka Python. Byly vybrány tři rámce napsané v jazyce Python: Django ORM, web2py DBAL a SQLAlchemy.

U každého rámce budou popsány techniky mapování, dotazování, funkce pro optimalizaci komunikace s databází a nevýhody rámce.

4.1 Django ORM

Django je rámec vytvořený v jazyce Python pro tvorbu webových aplikací. Je to jeden z nejpopulárnějších a nejpoužívanějších rámců pro tento jazyk [6]. Tvůrci tohoto rámce nazývají Django jako „webový framework pro perfekcionisty s termíny“. „Django podporuje rychlý vývoj a čisté, pragmatické konstrukce“ [7]. Jednoho z autorů tohoto rámce, Adriana Holovatyho, inspiroval kytarista Django Reinhard natolik, že po něm pojmenoval tento framework [8].

Django je známo hlavně pro své automaticky generované administrační rozhraní, obsáhlou dokumentaci a velkou komunitu uživatelů. Tento rámec se řídí návrhovým vzorem model-view-controller (MVC). ORM, které je součástí rámce představuje modelovou část.

ORM obsahuje i DBAL a lze jej tak využít nezávisle na konkrétním SŘBD. Mezi podporované SŘBD patří Oracle, MySQL, PostgreSQL a SQLite. Pro SQLite jsou však některé funkce omezeny a nelze tak v případě jeho použití využít celkový potenciál ORM. Je to však dáno i omezenou funkcí tohoto SŘBD ve srovnání s ostatními.

ORM lze používat nezávisle na rámci Django. V současné době je Django ve verzi 1.5. Následující text tak popisuje vlastnosti, které jsou obsaženy v této verzi.

4.1.1 Mapování v Django ORM

Objektově-relační mapování je v Djangu inspirováno návrhovým vzorem Active Record. Třída, která abstrahuje databázovou tabulku, tedy kromě mapování obsahuje i aplikační logiku. Tuto třídu Django pojmenovává jako „model“.

Aby třída získala vlastnosti modelu musí dědit z třídy `Model`. Při použití existující databáze lze definici této třídy automaticky vygenerovat pro každou tabulku. Django však preferuje přístup, při kterém se nejdříve vytvoří modely, a na základě této definice pak struktura v databázi.

Pro mapování sloupců používá model atributy, které jsou instancemi potomků třídy `Field`. Podporovány jsou pouze nejčastěji používané datové typy, což je dáno nezávislostí ORM na konkrétním SŘBD. Kromě těchto typů Django některé základní typy rozšiřuje a přidává k nim aplikační logiku, například atribut obsahující pouze emailové adresy.

Příkladem modelu je třída `Author` ve výpisu 3. Tento model je namapován na databázovou tabulku `blogapp_author`, kterou Django vygeneruje automaticky. Automaticky generován je i primární klíč, toto chování lze však změnit. Je ovšem vyžadováno, aby každá tabulka měla primární klíč. Dále pak musí primární klíč sestávat pouze z jednoho atributu, jelikož ORM nepodporuje složené klíče.

V definici modelu lze zadat i vytvoření indexu nebo unikátního integritního omezení pro konkrétní atribut. ORM indexy automaticky vytváří pro cizí klíče.

Django ORM podporuje i použití více databází najednou. Pro každý model je v takovém případě potřeba určit k jaké databázi náleží.

4.1.1.1 Instance modelu Instance modelu kromě dat řádku tabulky obsahují i metody pro práci s těmito daty. Mezi nejdůležitější patří metody pro validaci, uložení a mazání dat. Data se tedy před uložením validují i v ORM i v databázi. Toto má za důsledek mírně zvýšenou složitost, jelikož například při validování cizího klíče ORM kontroluje, zda-li v databázi existuje odkazovaný záznam.

Instance používají stejnou metodu pro vytvoření nového záznamu i pro uložení změn. Pokud instance neobsahuje primární klíč, je vytvořen nový záznam. Pokud klíč obsahuje, proběhne nejdříve kontrola, zda-li záznam se stejným primárním klíčem již existuje, a na základě výsledku se provede daný SQL příkaz.

Změny dat v instanci se v databázi projeví až při zavolání metody pro uložení.

4.1.1.2 Mapování relací Django ORM umožňuje definovat relace pro všechny způsoby popsané výše v sekci 3.2. Pro definování relace je potřeba využít tříd z balíku `django.db.models` a to `OneToOneField` pro relaci 1:1, `ForeignKey` pro 1:N nebo `ManyToManyField` pro N:M. Přístup k relačnímu objektu je pak realizován pomocí relačního atributu. Pro získání relace je proveden dotaz do databáze.

K relaci lze přistoupit i přes tzv. zpětnou relaci, kdy je možné se z odkazovaného objektu dostat na odkazující objekt. V případě relace 1:N nebo N:M je možné relaci dále filtrovat pomocí stejných pravidel jako při dotazování.

Při použití relace N:M je automaticky vytvořena vazební tabulka. Pokud je to potřeba, lze tuto tabulku předefinovat tak, aby obsahovala doplňkové informace o vazbě. Ve výpisu 14 je mezi modely `Author` a `Entry` automaticky vytvořena vazební tabulka.

Je možno vytvořit i rekurzivní relaci, relaci na objekty stejného typu.

4.1.1.3 Mapování děčnosti ORM rámce Django podporuje dva typy z výše popsaných návrhových vzorů mapování děčnosti v sekci 3.3. `Concrete Table Inheritance` a `Class Table Inheritance`. Oba dva případy jsou použity ve výpisu 14.

Vzoru `Concrete Table Inheritance` využívá model `Author`, který dědí z abstraktního modelu `Person`. Model `Person` je označen jako abstraktní pomocí meta třídy. Instance tohoto modelu nelze ukládat, slouží pouze pro společné definice modelů, které z něj dědí.

Modely `Entry` a `VlogEntry` využívají `Class Table Inheritance`. Mezi těmito modely ORM automaticky vytvoří relaci 1:1. Pokud je instance bazového modelu instancí i odvozeného modelu, je možné přistoupit přes relaci k datům odvozeného modelu. Při získávání dat z bazového modelu, ORM automaticky získá data spojením tabulek i z modelu odvozeného.

4.1.2 Dotazování

Django ORM používá pro dotazování do databáze třídy, které pojmenovává jako „manažery“. Každý manažer musí dědit z třídy `Manager` a každý model obsahuje jeden nebo více těchto manažerů. Implicitně lze manažer nalézt pod atributem modelu `objects`. Takováto struktura tak odděluje manipulaci s celou tabulkou a manipulaci s řádkem tabulky.

Manažer poskytuje aplikační programové rozhraní (API) pomocí kterého abstrahuje SQL. Toto rozhraní nemá za úkol nahradit veškeré SQL, pouze usnadnit provádění častých dotazů. ORM však jako celek instruuje svými strukturami snahu nevyjadřovat výrazy pomocí SQL.

API obsahuje metody pomocí kterých lze objekty filtrovat, agregovat, řadit, limitovat, a na pozadí provádí SQL. Každá z těchto metod vrací objekt typu `QuerySet`. `QuerySet` se stará o samotné provedení dotazu, získání dat z databáze a vytváření instancí modelů. `QuerySet` otálí s provedením dotazu, dokud data nejsou opravdu potřeba. Toto umožňuje jednotlivé metody pro dotazování řetězit. Je také možné `QuerySet` uložit do proměnné a vždy získat aktuální data, jelikož při řetězení metod je pokaždé vrácen nový `QuerySet`.

Mezi nejdůležitější metody objektu `QuerySet` patří `all`, `filter`, `exclude` a `get`. Metoda `all` vrací všechny objekty daného modelu, `filter` jenom objekty, které splňují podmínku, `exclude` objekty, které podmínku nesplňují a `get` vrací jediný objekt pro danou podmínku. `QuerySet` je možné dále omezovat pomocí standardní konstrukce používané pro objekty typu pole v jazyce Python. ORM poskytuje i možnost zadat vlastní SQL, z jehož výsledku se pak vytvoří instance konkrétního modelu.

Po každé operaci Django ORM provádí implicitně `commit`. Pokud je ORM použito nezávisle na rámci, nejsou transakce podporovány.

API pro dotazování poskytuje mnoho dalších funkcí, jejich popis je však nad rámec této diplomové práce. Další informace lze nalézt v oficiální dokumentaci [9]. Ukázku API rámce Django ORM pro práci z daty zobrazuje výpis 6 v sekci 4.4.2.

4.1.3 Optimalizace v Django ORM

Django ORM umožňuje svými funkcemi optimalizovat některé náročnější operace. Přihlédnutím k těmto optimalizacím lze ve většině případů zlepšit výkon ORM.

4.1.3.1 Ukládání do mezipaměti Pro každý provedený dotaz ORM automaticky kešuje výsledná data. Data nejsou uložena ve společné keši, ale ve výsledném objektu `QuerySet` nebo v konkrétní instanci. Jelikož však ORM při řetězení metod vrací vždy nový `QuerySet`, keš je při každé změně dotazu zahozena a data jsou načtena znovu.

Keš je použita v instanci v případě přístupu k relacím. Platí to však pouze, pokud relace odkazuje na konkrétní objekt. Keš se u relací M:N nebo zpětných relací u instance nepoužije [10].

V případech, kdy výsledný `QuerySet` obsahuje mnoho záznamů, lze vypnutím této keše předejít zahlcení paměti [11].

4.1.3.2 Načítání relací Při neopatrném používání ORM může dojít k tzv. *N+1 problému*. N+1 problém nastává, pokud je použit cyklus při získávání dat z databáze spolu s líným nahráváním. Většinou se tento problém projevuje u relací.

Výpis 1 používá schéma definované v sekci 4.4. Funkce `inefficient` v tomto výpisu zobrazuje, jak N+1 problém vypadá. Funkce se stará o jednoduchý výpis příspěvků a blogů, ve kterém jsou publikovány. Tato funkce však provede N+1 dotazů do tabáze. Jeden dotaz pro získání všech příspěvků a N dotazů pro získání blogu každého příspěvku. Tento kód navíc nebere v potaz ani to, že některé příspěvky mohou být publikovány ve stejném blogu.

ORM poskytuje řešení v podobě funkcí `select_related` a `prefetch_related`.

`select_related` se používá při přednačtení relací, které odkazují na konkrétní objekt. Při použití této funkce se všechny výsledné objekty a jejich relace načtou jedním dotazem pomocí spojení tabulek, a relační objekty se uloží do keše k dané instanci. Funkce `efficient` ve výpisu 1 tedy použije pouze jeden dotaz do databáze.

Stejný problém může nastat i při použití zpětných relací nebo v případech, kdy relace odkazuje na kolekci objektů. Pro tento případ se využívá metoda `prefetch_related`. Funkce `efficient_backward` ve výpisu 1 využívá této metody. Při spuštění se provedou dva dotazy. Jeden pro získání všech blogů a druhý pro získání všech příspěvků. Výsledek se opět uloží do keše k daným instancím blogu.

```
def inefficient():
    for entry in Entry.objects.all():
        print 'Příspěvek: ', entry.headline
        print 'Publikován v blogu: ', entry.blog.name #Je proveden
            dotaz do databáze

def efficient():
    for entry in Entry.objects.select_related('blog').all():
        print 'Příspěvek: ', entry.headline
        print 'Publikován v blogu: ', entry.blog.name #Hodnota je zís
            skána z keše

def efficient_backward():
    for blog in Blog.objects.all().prefetch_related('entry_set'):
        print 'Blog: ', blog.name
        print 'Příspěvky v blogu: '
        for entry in blog.entry_set.all(): #Hodnota je získána z keše
            print entry.headline
```

Výpis 1: N+1 problém a jeho řešení v Django ORM

4.1.3.3 Omezení sloupců Implicitně se ORM pro každou tabulku dotazuje na všechny její atributy. Tento přístup může být obzvlášť náročný při obsáhlých tabulkách nebo objemných atributech, proto je možné dotazy omezit na konkrétní atributy tabulky. Tento postup se však doporučuje používat pouze, pokud není jisté, které konkrétní atributy

budou použity. V opačném případě se doporučuje tabulku rozdělit, jelikož tato funkce mírně zvyšuje režijní náklady ORM [11].

4.1.3.4 Hromadné operace ORM umožňuje pro objekty stejného typu hromadné operace. Mezi poskytované funkce patří hromadné vytváření, mazání i ukládání. Ve všech případech je však generován vždy jen jeden dotaz a ORM tak neumožňuje zpracovávání po dávkách. Další nevýhodou je, že při provedení těchto operací je přeskočena aplikační logika v modelech [10].

4.1.3.5 Ukládání změn Od verze 1.5 Django ORM umožňuje při ukládání instance uložit pouze specifikované atributy. Je tak možno uložit pouze změněná data a nezatěžovat databázi odesíláním všech atributů, což se v případě obsáhlého modelu může pozitivně projevit na výkonu. ORM však neposkytuje žádný mechanismus, který by tento přístup automatizoval, je tak potřeba implementovat vlastní sledování změn [12].

4.1.3.6 Nevytváření instancí ORM poskytuje možnost vyhnout se vytváření instancí modelu z dat vrácených databází. Jelikož je proces vytváření instancí náročný, může tento postup v případech, kdy je potřeba získat pouze data, zlepšit celkový výkon [11].

4.1.3.7 Zjišťování existence Pro zjišťování existence záznamů v databázi umožňuje ORM použít úsporný dotaz `SELECT (1) . . .`. Využití této funkce je tedy většinou výhodnější než porovnávání výsledku obyčejného dotazu [11].

4.1.4 Nevýhody Django ORM

Django ORM představuje jednoduchou jednovrstvou abstrakci databáze poskytující O/R mapování. Má ovšem i své nevýhody.

Některé z nevýhod vychází z použití návrhového vzoru Active Record. Tento fakt předurčuje Django ORM pro použití v případech jednoduchého relačního modelu v databázi. U složitějších konstrukcí se projeví nevýhody tohoto vzoru popsané v sekci 3.1.3.

ORM je také zaměřeno více na jednoduchost mapování a aplikační logiky a provádí ústupky na úkor databáze. Při použití jednoduchých aplikací lze toto přehlédnout, avšak v případech systémů, kdy databáze přetrvávají déle než aplikační kód, je toto řešení nevyhovující.

Další nevýhody plynou z použití O/R mapování. API ORM zakrývá skutečné dotazy prováděné nad databází, což může vést k neuváženému používání a zatěžování databáze. Je proto vždy nutné vědět, co ORM v konkrétních případech provádí. Toto má však za následek zdvojnásobení znalostí potřebných k naučení, jelikož je potřeba znát SQL i ORM API. Opět lze tento problém přehlédnout v případě jednoduchých aplikací, u složitějších aplikací je to však potřeba.

Mezi hlavní nevýhody pak patří nedostatečná abstrakce a inkonzistence v syntaxi API pro dotazování do databáze. ORM obsahuje dva typy konstrukcí pro získávání objektů

z databáze. Tyto dva typy jsou vzájemně nekompatibilní, což způsobuje, že i poměrně jednoduché dotazy nelze pomocí API provést. Příkladem je dotaz, který pro každý blog vypíše celkový čas strávený modifikací jeho příspěvků. Výpis 2 zobrazuje SQL dotaz pro získání těchto dat. Pomocí API ORM však nelze vytvořit adekvátní protějšek tohoto SQL [13].

```
SELECT blog_id, SUM(mod_date – pub_date) FROM blogapp_entry GROUP BY blog_id;
```

Výpis 2: Příklad dotazu, který nelze provést pomocí Django ORM API

ORM umožňuje zadat vlastní SQL dotazy, což řeší většinu problémů. Toto ovšem není dokonalé řešení, jelikož jsou tak ztraceny některé výhody ORM. V případě použití vlastního SQL se ztrácí možnost nezávislosti na konkrétní databázi, modelové schéma v aplikaci se rozpadá a začlenění dodatečné logiky přináší komplikace.

4.1.5 Zhodnocení

Navzdory nevýhodám uvedeným výše Django ORM funguje skvěle pro jednoduché případy a jeho použití ušetří mnoho času. A protože se jedná o projekt, který se každým dnem vyvíjí, je možné, že budou některé z výše popsanych nevýhod v budoucnu odstraněny.

4.2 web2py DBAL

web2py je rámec pro tvorbu webových aplikací napsaný v jazyce Python, který vznikl původně jako univerzitní projekt vytvořený profesorem Massimem Di Peirrem pro výukové účely [14]. Byl stvořen s úmyslem co nejvíce usnadnit vývoj webových aplikací a problémy s tím spojené. Celý rámec je tak distribuován pomocí balíku, který v základu obsahuje integrované vývojové prostředí (IDE), webový server, relační databázi SQLite a dokonce i interpret jazyka Python. Není tedy potřeba nic instalovat, balík se rozbalí a ihned lze tento rámec používat [15]. Rámec se řídí návrhovým vzorem MVC.

IDE sestává z administračního rozhraní, které je spuštěno na webovém serveru. Kód lze tak měnit v tomto IDE přímo na serveru a změny jsou viditelné ihned. Rámec obsahuje pro vytvořené aplikace i automaticky generované administrační rozhraní nad daty podobně jako Django.

web2py implementuje DBAL pro komunikaci s databází. Odlišnost této vrstvy od ORM je popisována jako „mapování databázových tabulek a záznamů na instance a třídy reprezentující množiny a záznamy namísto vysoko-úrovňové abstrakce“ [14]. Mezi podporované SŘBD podle oficiální dokumentace patří Firebird, IBM DB2, Informix, Ingres, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, Google App Engine a mnoho dalších. Stejně jako Django ORM lze tuto vrstvu používat nezávisle na rámci.

4.2.1 Mapování ve web2py DBAL

Třída `web2py.gluon.dal.DAL` je vstupním bodem pro komunikaci s databází. Inicializací objektu této třídy je vytvořeno spojení s databází. DBAL obsahuje connection pool, do kterého se spojení uloží. Connection pool je možno konfigurovat podle potřeby.

Rámec vyžaduje pro správnou funkčnost datové vrstvy nejdříve definovat strukturu tabulek v databázi. Definice struktury slouží jako metadata při dotazování do databáze a umožňuje tabulky v databázi automaticky generovat i měnit. Definice tabulek si DBAL ukládá do speciálních souborů. Pomocí těchto souborů lze pak provádět migrace databáze.

Migracemi lze měnit datové typy konkrétních sloupců, odstraňovat nebo přidávat nové sloupce k tabulkám a nebo i přesouvat celé databáze do jiných SŘBD. Migrace však nejsou úplně automatizované a mohou při nich nastat problémy, které je potřeba řešit explicitně. Kromě migrací umožňuje DBAL i export dat do XML, HTML nebo CSV formátu.

DBAL umožňuje používat i existující databázi, je však potřeba, aby splňovala určité podmínky. Konkrétně musí databáze obsahovat pro každou tabulku primární klíč a všechny relace musí být řešené pomocí primárního klíče. web2py DBAL podporuje složené klíče.

Tabulky lze definovat pomocí instance třídy `DAL`. Je také možné pomocí tohoto objektu tabulkám dále nastavovat další parametry, měnit jejich strukturu nebo je i smazat. Primární klíč je z definice automaticky vygenerován pro každou tabulku. Sloupce definice tabulky pak obsahují metody pro validaci, které lze využít před ukládáním do databáze. Výpis 4 zobrazuje definici tabulky `author` využívající abstraktní strukturu `person`.

DBAL neumožňuje pomocí této definice určit databázové indexy.

4.2.1.1 Práce se řádkem tabulky Každý dotaz do databáze vrací množinu objektů třídy `Row`. Tento objekt nepředstavuje záznam tabulky, ale řádek výsledku vráceného na dotaz. Data je možné získat přes atributy řádku, a pokud řádek obsahuje primární klíč, lze s řádkem dále pracovat, modifikovat jeho data nebo řádek v databázi smazat.

Nová data se neukládají pomocí řádku, ale pomocí objektu tabulky. Řádek však obsahuje metodu, která automaticky zjistí, zda-li řádek v databázi vytvořit nebo aktualizovat novými daty. Tato metoda se nemusí řídit pouze pomocí primárního klíče jako v případě Django ORM, ale lze určit atributy, podle kterých metoda provede buď vytvoření nebo aktualizaci.

4.2.1.2 Mapování relací DBAL podporuje implicitně pouze relace 1:N. Relační záznam lze získat přes atribut řádku, který obsahuje cizí klíč. DBAL provede dotaz do databáze a výsledek je k tomuto atributu přiřazen.

K relaci lze přistoupit i z odkazovaného objektu. Pro realizaci zpětné relace je potřeba přistoupit přes atribut řádku jehož název je totožný s názvem tabulky, do které náleží vazební objekty. Vazební objekty je pak možné získat všechny nebo jejich množinu dále limitovat stejným způsobem jako při dotazování.

Jak je možné vidět ve výpisu 15, pro relaci N:M mezi strukturami `author` a `entry` bylo potřeba definovat vazební tabulku. DBAL automaticky vazbu N:M nevytváří ani nepodporuje. Pro získání vazebních objektů je v případě N:M potřeba explicitně provést spojení tabulek.

Rekurzivní relace pomocí DBAL definovat lze.

4.2.1.3 Mapování dědičnosti Datová vrstva web2py neumožňuje mapování dědičnosti, jelikož neobsahuje ORM. Nejjednodušší z návrhových vzorů pro mapování dědičnosti, Concrete Table Inheritance, lze ovšem nasimulovat pomocí struktur. Příklad zobrazuje výpis 15, kde tabulka `author` rozšiřuje abstraktní definici `person`. Tabulka `person` se v databázi neobjeví a tabulka `author` obsahuje všechny své definované sloupce i sloupce z definice `person`. V tomto výpisu je dále definována tabulka `vlog_entry`, která rozšiřuje konkrétní definici `entry`. Obě tyto tabulky jsou v databázi vytvořeny.

Tabulka `vlog_entry` pak obsahuje všechny sloupce definované ve své definici a navíc i všechny sloupce tabulky `entry`.

4.2.2 Dotazování

Pro dotazování do databáze se používá stejný objekt jako pro práci s tabulkami. Tento objekt přijímá jako parametr podmínku a vrací instanci třídy `Set`, která poskytuje metodu `select`. Metoda `select` pak přijímá široké spektrum atributů, kterými lze dotaz dále filtrovat a limitovat. Výsledkem dotazu jsou objekty řádků, jak bylo popsáno v sekci 4.2.1.1.

Při tvoření dotazů pomocí API se nemusí specifikovat tabulky, protože DBAL dokáže automaticky zjistit potřebné atributy z ostatních parametrů předaných do dotazu. To platí i u složitějších dotazů jako je například spojení tabulek.

Výsledná data získaná z databáze lze dále filtrovat a to bez potřeby nového dotazu do databáze. DBAL k těmto účelům poskytuje speciální API, které pracuje s výslednými daty v paměti.

DBAL umožňuje pokládat dotazy i za pomoci vlastně vytvořeného SQL. Takto sestavený dotaz však nevrací data ve formě instancí třídy `Row`, ale v základní formě, kterou vrací konkrétní databázový ovladač.

Transakce jsou v DBAL automaticky započaty, jakmile jsou vyžadovány nějaké operace nad databází. Po vykonání potřebných operací je potřeba vždy explicitně provést `commit` nebo `rollback`. Ukázkou API rámce web2py DBAL pro práci z daty zobrazuje výpis 7 v sekci 4.4.2.

4.2.3 Optimalizace ve web2py DBAL

DBAL neposkytuje mnoho optimalizací pro dotazování do databáze, jelikož syntaxe API je velmi podobná SQL a je tak možné pomocí tohoto API vytvořit skoro jakýkoliv SQL dotaz.

Problém N+1 DBAL nijak neulehčuje, a je ho potřeba řešit explicitním spojením tabulek. Jelikož je však ze syntaxe skoro jasné jaké dotazy se provádí, lze se tomuto problému jednodušeji vyhnout.

Mezi optimalizace, které DBAL poskytuje patří kešování výsledku dotazů a hromadné operace pro ukládání dat.

4.2.4 Nevýhody web2py DBAL

web2py DBAL poskytuje velmi dobré API pro dotazování do databáze. Tento fakt však zastiňuje mnoho nevýhod.

Vytváření nebo změny tabulek jsou bohužel prováděny za běhu aplikace. DBAL také neulehčuje použití existující databáze, jelikož neposkytuje nástroj, který by automaticky vygeneroval definice tabulek a je tak potřeba tyto definice vytvořit ručně.

Jelikož jsou metadata databázové struktury tabulek používány pro dotazování, není možné se bez těchto definic obejít, i pokud není potřeba vytvářet nebo měnit databázové tabulky. Pouze pokud již má rámec uložené metadata ve svých souborech, lze se definicím vyhnout [16]. Tyto metadata však vznikají z definic tabulek, proto nelze použít DBAL jen jako dotazovací nástroj.

Další nevýhodou DBAL je, že jeho databázová abstrakce je nedostatečná. Přbytek funkcí, které obsahuje nelze použít pro všechny SŘBD a mnoho SŘBD lze používat pouze s omezením. Příkladem jsou některé datové typy, které jsou nefunkční pro polovinu podporovaných SŘBD. DBAL tedy spoléhá hodně na fakt, že se SŘBD nebude měnit.

Největší nevýhodu však vidím v návrhu této vrstvy. Obsahuje mnoho návrhových chyb, které se můžou projevit negativně na projektech vytvořených pomocí této vrstvy a dokonce i na stabilitě samotné vrstvy. Všechny instance tříd na datové vrstvě obsahují referenci na spojení do databáze. DBAL míchá dohromady funkce spadající na datovou vrstvu s funkcemi, které spadají do jiných vrstev a obsahuje mnoho nepotřebných funkcí, které by měly být distribuovány spíše pomocí přídatných modulů.

Navíc je samotná struktura kódu DBAL velice nepěkná. Skládá se z jednoho souboru o 10000 řádcích.

4.2.5 Zhodnocení

Rámec web2py vznikl jako školní projekt pro výukové účely. Bohužel tento fakt a nevýhody z toho plynoucí jsou stále znát na kvalitě celého rámce i přesto, že od doby vzniku prošel mnoha změnami.

Mezi jeho výhody patří dotazovací API, které má blízko k SQL, funkce jako migrace a verzování databázových záznamů. Jeho využití je pro ukázkové nebo velmi jednoduché projekty. Pro seriózní projekty se web2py DBAL však kvůli výše uvedeným nevýhodám nevyplatí používat.

4.3 SQLAlchemy

SQLAlchemy je rámec pracující na datové vrstvě velmi orientovaný na SQL. Rámec byl vytvořen s myšlenkou, že „relační databáze se chová méně jako kolekce objektů, jakmile začne více záležet na efektivitě a výkonu, a naopak kolekce objektů se chovají méně jako tabulky a záznamy, jakmile záleží více na abstrakci databáze“[27]. Rámec tak umožňuje vždy přistoupit přímo k databázi, pokud je to nutné a zároveň poskytuje velmi dobrou abstrakci, která umožňuje kompletní odstínění aplikační logiky od specifik databáze.

SQLAlchemy bylo inspirováno rámcem Hibernate vytvořeným v programovacím jazyce Java a stejně jako tento rámec se řídí návrhovým vzorem Data Mapper.

Rámec původně vznikl pouze jako generátor dotazů a postupně, jak se vyvíjel, byly přidávány další vrstvy, vlastnosti a funkce. Obrázek 6 zobrazuje architekturu rámce tak, jak vypadá ve verzi 0.8.

Mezi dvě hlavní vrstvy rámce patří jádro a ORM. Jádro SQLAlchemy se skládá z DBAL a je možné ho využít samostatně bez ORM. DBAL obsahuje moduly Dialect, Connection Pooling, Engine, SQL Expression Language a Schema.

Dialect Modul Dialect obsahuje specifické vlastnosti pro každý podporovaný SŘBD a databázový ovladač. SQLAlchemy se snaží kromě mnoha SŘBD podporovat i různé typy ovladačů pro každý SŘBD. Tento modul tedy zpracovává požadavky a aplikuje je pro konkrétní SŘBD a použitý ovladač.

Connection Pooling Connection Pooling je standardní technikou používanou pro správu spojení, která efektivně hospodáří s připojením do databáze a poskytuje možnost používat spojení na více místech současně. Tento modul umožňuje nastavit vlastnosti a chování této funkce jako je například obnova přerušených spojení nebo množství spravovaných spojení.

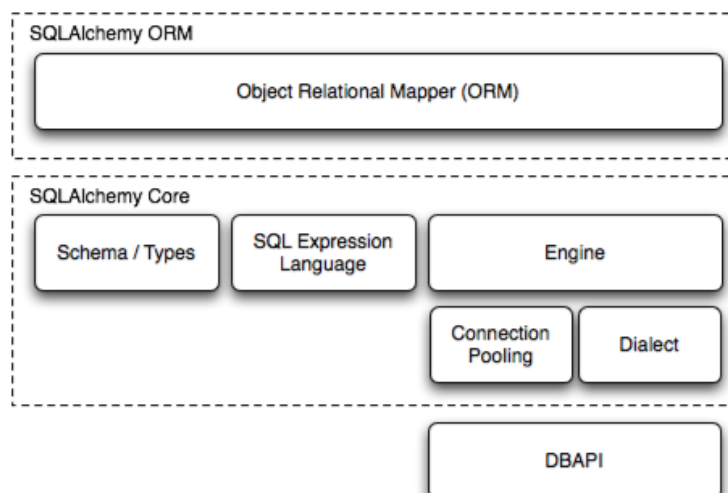
Engine Modul Engine se stará o správu konkrétního spojení. Získává spojení z modulu Connection Pooling, zakládá a uzavírá transakce, předává požadavky modulu Dialect a loguje příkazy spuštěné nad databází. Tento modul je hlavním rozhraním k databázi.

SQL Expression Language Modul SQL Expression Language obsahuje API pro generování SQL dotazů. API je velmi variabilní a poskytuje mnoho možností, jak dotazy tvořit. Umožňuje například vytváření vnořených dotazů, používání databázových funkcí a procedur, skládání dotazů zčásti za pomoci API a zčásti ručně vytvořeným SQL nebo i jenom pomocí vlastního SQL.

Při použití tohoto API je jasné vidět jaké SQL dotazy se provádí a přitom je umožněno dotazy vytvářet jednoduše a přehledně.

Schema/Types V tomto modulu jsou obsaženy funkce pro definování struktury a mapování relačního schématu v databázi. Pomocí tohoto modulu je možné nadefinovat databázové tabulky, jejich atributy, vlastnosti a integritní omezení. Rámec umožňuje definovat datové typy pomocí datových typů jazyka Python i pomocí datových typů relační databáze. Tyto typy jsou pak mezi sebou mapovány co nejpřesněji. SQLAlchemy podporuje mnoho databázových datových typů a umožňuje nadefinovat i vlastní.

Při použití ORM rámce SQLAlchemy je možné v místech, kde ORM nevystačuje využít DBAL a SQL Expression Language. DBAL rámce SQLAlchemy je velice podobná DBAL rámci web2py a nebude zde tedy podrobně popsána. Následující popis se tak zaměřuje na ORM.



Obrázek 6: Architektura rámce SQLAlchemy [28, str. 3]

Mezi SŘBD podporované rámcem patří Oracle, MSSQL Server, MySQL, PostgreSQL, SQLite, Sybase a mnoho dalších. Přes přídatné balíčky lze získat podporu i pro další SŘBD.

4.3.1 Mapování v SQLAlchemy

Jak bylo popsáno výše, SQLAlchemy ORM se při mapování řídí návrhovým vzorem Data Mapper. Mapování se tedy skládá ze dvou struktur, a to z třídy z modulu Schema, která představuje strukturu databázové tabulky a z mapovací třídy, která představuje doménový objekt. Definice tříd představující struktury v databázi slouží jako metadata a používají se při vytváření dotazů do databáze. Mapovací třída je typu POPO (Plain Old Python Object), nerozšiřuje tedy žádnou básovou třídu a neimplementuje žádné rozhraní. V mapovací třídě lze vytvořit i validátory dat, které je možné volat před uložením dat do databáze.

Pomocí tříd, které představují strukturu databázových tabulek umožňuje SQLAlchemy vygenerovat SQL kód k jejich vytvoření a při použití existující databáze lze vygenerovat pomocí introspekce z databázových tabulek definici těchto tříd.

SQLAlchemy lze bez problému používat s existující databází. Jedinou podmínkou je, aby každá tabulka obsahovala primární klíč. Rámec dovoluje používat složené klíče i složené vazební atributy. SQLAlchemy ovšem nepodporuje migrace databázových dat. Této funkce lze dosáhnout až přes zásuvné moduly třetích stran.

Vzhledem k použití návrhového vzoru Data Mapper obsahuje SQLAlchemy mnoho možností, jak namapovat struktury v databázi na doménové objekty. Na jeden doménový objekt je možné namapovat více tabulek nebo část tabulky. Lze vytvořit hybridní a složené

atributy, kompozitní sloupce nebo namapovat doménový objekt či jeho atribut přímo na SQL dotaz.

SQLAlchemy ORM obsahuje i možnost definovat schéma a struktury v databázi podobně jako při použití návrhového vzoru Active Record. Pomocí tzv. deklarativního přístupu lze nadefinovat za pomoci jediné třídy mapovací třídu i třídu představující strukturu v databázi. Příkladem je třída `Author` ve výpisu 5.

4.3.1.1 Mapování relací Rámec umožňuje definovat všechny typy relací popsaných v sekci 3.2. Při definování relací je třeba rozlišovat mezi relacemi na úrovni databáze a relacemi mezi objekty v doménové logice. Obě tyto relace je potřeba specifikovat. Při definici relací mezi doménovými objekty je možné určit zpětné relace. Dále je možné určit jakým způsobem se budou vazební objekty načítat, zdali se budou načítat všechny a zdali je bude možné dále filtrovat.

Pro vytvoření relace N:M je potřeba vždy pomocí modulu `Schema` definovat vazební tabulku. SQLAlchemy pak automaticky zajistí načítání objektů přes tuto vazební tabulku při přístupu k N:M relaci a poskytuje i možnost jednoduchého vytvoření relačních objektů. Rámec podporuje i rekurzivní relace.

Před úpravou nebo mazáním relačních objektů rámec vždy tyto objekty z databáze načte. Při mazání relačních objektů navíc implicitně nepředpokládá, že je v databázi nastaveno kaskádové mazání a je potřeba to explicitně zadat nebo vazební objekty smazat samostatně.

4.3.1.2 Mapování dědičnosti SQLAlchemy podporuje všechny tři typy dědičnosti popsaných v sekci 3.3. Rámec se řídí návrhovými vzory dědičnosti velmi striktně a implementuje dědičnost přesně podle definicí.

Bázová třída řídicí se vzorem `Class Table Inheritance` obsahuje atribut, který určuje identitu objektů v hierarchii. Každá třída v hierarchii pak obsahuje definici této identity a konkrétní třídy obsahují navíc cizí klíč na báзовou tabulku. Primární klíče jsou uloženy pouze v báзовé tabulce. Rámec umožňuje získat objekty z těchto tabulek pomocí dotazů, které využívají polymorfismus.

Podobným způsobem lze definovat i dědičnost podle vzoru `Single Table Inheritance`. Báзовá třída má opět atribut, který určuje identitu objektů a každá třída v hierarchii definici identity. Konkrétní tabulky pak již neobsahují cizí klíč na báзовou třídu, jelikož celá hierarchie je reprezentována v databázi jedinou tabulkou.

Při definici tříd, které využívají `Concrete Table Inheritance` je potřeba, aby abstraktní báзовá třída dědila z třídy `AbstractConcreteBase`. U konkrétních tříd je pak potřeba nastavit atribut, který určuje vytvoření tabulky v databázi, a definovat jejich identitu v rámci hierarchie. U tříd, u kterých není definovaná identita, nelze použít načítání objektů z databáze využívající polymorfismus.

4.3.2 Dotazování

SQLAlchemy používá pro práci s objekty systém řídicí se návrhovým vzorem *Unit of Work* [1]. Unit of Work představuje mechanismus, který synchronizuje všechny změny mezi objekty a jejich vazební záznamy. Tento systém v SQLAlchemy zaštiťuje objekt třídy *Session* neboli „sezení“.

Objekt třídy *Session* se stará o vytváření transakce skrze modul *Engine*, o stavy všech objektů registrovaných v sezení, a provádí SQL dotazy ve chvílích, kdy je to potřeba. Sezení otálí s prováděním dotazů do doby než je to nevyhnutelně nutné. Objekty registrované v sezení se pohybují mezi stavy *transient*, *pending*, *persistent* nebo *detached*.

Stav *transient* Objekt ve stavu *transient* je instance mapovací třídy, která nebyla zatím registrována v sezení a data, která obsahuje nebyla uložena do databáze.

Stav *pending* Instance, která byla registrována do sezení, ale její data zatím nebyla uložena do databáze, je ve stavu *pending*.

Stav *persistent* Objekty, které jsou získány z databáze nebo instance, jejichž data byla uložena do databáze jsou ve stavu *persistent*.

Stav *detached* Instance, která obsahuje data z databáze, ale nebyla registrována do sezení, se nachází ve stavu *detached*. Instanci v tomto stavu lze používat, ale nelze načítat její vazební objekty z databáze.

Pokud je objekt zaregistrovaný v sezení aktualizován, sezení tuto změnu automaticky zaznamená a v příhodné chvíli provede změny v databázi. Všechny změny, které má sezení zaregistrováno jsou provedeny v případě, že je přes sezení vyžadováno načtení nového objektu z databáze, a nebo pokud je transakce ukončena příkazem `commit` nebo `rollback`. Sezení po ukončení transakce automaticky započne novou transakci, jakmile jsou přes sezení vyžadovány další manipulace s objekty. SQLAlchemy umožňuje vytvářet pomocí sezení i vnořené transakce.

Při vytváření dotazů do databáze vrací sezení instanci třídy *Query*. Tento objekt poskytuje metody pro získávání objektů z databáze. Metody umožňují objekty filtrovat, limitovat a řadit. Výsledky je možné získat ve formě instancí mapovacích tříd nebo v objektech typu pole. API umožňuje při získávání objektů z databáze zadat části dotazu ve formě vlastně vytvořeného SQL.

SQLAlchemy ORM využívá při práci s objekty strukturu Identity Map. Při získávání objektů z databáze tato struktura zjistí, zda je objekt se stejnými daty v sezení již registrován, a pokud ano vrací pouze referenci na tento objekt. Funguje také zčásti jako jednoduchá keš, která obsahuje objekty rozdělené podle jejich unikátního identifikátoru. Objekty jsou v Identity Map registrovány pouze po dobu transakce. Jakmile transakce skončí, data jsou označena jako zastaralá a všechna data objektů v paměti je potřeba znovu načíst z databáze.

Ukázku API rámce SQLAlchemy pro práci z daty zobrazuje výpis 8 v sekci 4.4.2.

4.3.3 Optimalizace v SQLAlchemy

SQLAlchemy provádí mnoho optimalizací na pozadí. Jeho velkou výhodou je použití sezení, které umožňuje implicitně provádět hromadné operace a spolu s Identity Map slouží jako jednoduchá keš pro data.

Mezi optimalizace, které je potřeba určit explicitně patří načítání relačních objektů.

4.3.3.1 Načítání relací Podobně jako Django ORM poskytuje SQLAlchemy optimalizace v podobě horlivého načítání relací při vytváření dotazů do databáze. Tyto optimalizace zabraňují, aby nastal N+1 problém, umožňují efektivní provedení SQL dotazů a vylepšují celkový výkon rámce.

SQLAlchemy umožňuje horlivého načítání dosáhnout pomocí vnořeného dotazu metodou `subqueryload`, implicitním spojením tabulek metodou `joinedload` a nebo explicitním spojením tabulek.

SQLAlchemy doporučuje vnořený dotaz používat při načítání kolekce vazebních objektů a dotaz využívající spojením tabulek při relaci N:1 [28]. Při spojením tabulek se totiž většinou přenáší více dat (jak bude popsáno v sekci 5.3.3) a pro získání kolekce vazebních objektů nemusí být spojením tabulek vždy nejefektivnějším řešením.

4.3.3.2 Načítání relací při aktualizaci dat Relační objekty je potřeba načíst z databáze před jejich smazáním nebo úpravou. SQLAlchemy provádí toto načítání na pozadí. V případě velkých objemů dat se tento fakt může negativně projevit na výkonu. Nastavením atributu `passive_deletes` a `cascade` v mapovací třídě, lze toto chování změnit.

4.3.4 Nevýhody SQLAlchemy

SQLAlchemy je velice sofistikovaným řešením. Jeho abstrakce je dostačující a v případech, kdy nevystačuje lze použít vestavěnou DBAL. Umožňuje získat z databáze maximální výkon, pokud je to potřeba, a nebo ji dostatečně abstrahovat.

Mezi nevýhody SQLAlchemy však patří jeho složitost, obsáhlost a nutnost vše explicitně nastavovat. Obsahuje mnoho funkcí, které nemusí být vždy využívány, mnoho možností a nastavení. Pro dokonalé porozumění celého rámce je před jeho použitím potřeba nastudovat obsáhlý manuál³. Rámec tedy vyžaduje velkou časovou investici, což nemusí být vždy výhodné.

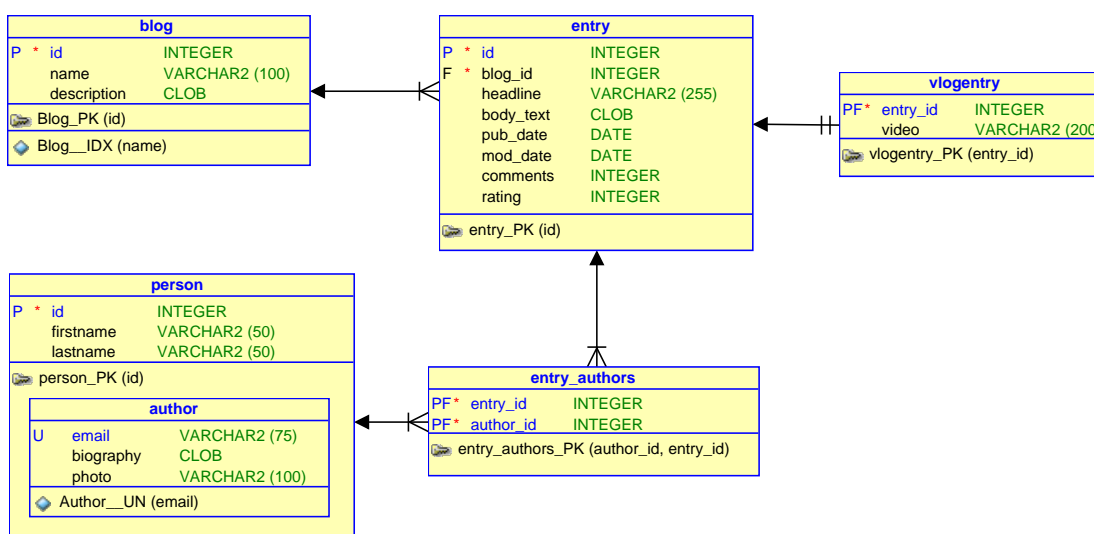
4.3.5 Zhodnocení

SQLAlchemy je nejvíce vyspělým rámcem z popisových technologií v této kapitole. Umožňuje vytvořit datovou vrstvu na míru konkrétním podmínkám, lze ho velmi dobře integrovat do existujících systémů a obsahuje podporu pro mnoho SŘBD. Jeho složitost, množství znalostí potřebných pro jeho efektivní používání a použití návrhového vzoru Data Mapper jej však činí vhodným většinou pouze pro komplexnější projekty.

³Dokumentace SQLAlchemy 0.8 dosahuje téměř tisíce stránek

4.4 Praktické srovnání existujících technologií pracujících na datové vrstvě

Pro praktické srovnání rámců popsaných v této kapitole bude každý z nich pracovat s databázovou strukturou, kterou zobrazuje ERD diagram na obrázku 7. Tato struktura představuje jednoduchou aplikaci pro vytváření blogů. Pokud to bude možné, budou se rámce snažit namodelovat tuto strukturu pomocí mapování. Konkrétně pak bude mezi strukturami `entry` a `vlogetry` využito dědičnosti Class Table Inheritance a struktura `author` bude dědit pomocí Concrete Table Inheritance z abstraktní struktury `person` obsahující atributy jméno a příjmení.



Obrázek 7: ERD diagram jednoduché aplikace pro vytváření blogů

Výpisy v sekci 4.4.1 zobrazují mapování entit `Person` a `Author` pro rámce Django ORM, web2py DBAL a SQLAlchemy. Kompletní mapování struktur z diagramu na obrázku 7 byla pro svoji délku vložena do přílohy A. Reprezentace těchto struktur v databázi vygenerovaná rámci z těchto mapování je pak zobrazena v příloze B.

Výpisy v sekci 4.4.2 zobrazují ukázkou práci s daty v rámci Django ORM, web2py DBAL a SQLAlchemy. Výpisy jsou ve formátu příkazové řádky, kdy řádek začínající znakem `>>>` reprezentuje operace, které je možné zadat na příkazové řádce jazyka Python, následované výstupem, které tyto operace vyvolaly. Ve výpisech jsou také přidány SQL dotazy, které jsou vygenerovány rámci. Tyto dotazy jsou vypsané na místech, kdy jsou odeslány do databáze.

Všechny výpisy vykonávají stejné operace. Nejdříve vytvoří nový záznam v tabulce `blog`. Poté tento záznam aktualizují a získají z databáze jeho vazební objekty ve formě záznamů tabulky `entry`. Dále je pak vytvořen dotaz do databáze na první záznam tabulky `entry`, jehož atribut `rating` je větší než nula a jehož atribut `headline` je první v abecedním pořadí. Nakonec jsou získány všechny záznamy tabulky `blog` seřazené podle atributu `name` pomocí vlastně vytvořeného SQL.

4.4.1 Mapování

```
class Person(models.Model):
    firstname = models.CharField(max_length=50)
    lastname = models.CharField(max_length=50)

    class Meta:
        abstract = True

class Author(Person):
    email = models.EmailField(unique=True)
    biography = models.TextField()
    photo = models.ImageField(upload_to="photos/")
```

Výpis 3: Ukázka mapování v Django ORM

```
db = DAL("postgres://username:password@localhost/database")

person = db.Table(db, 'person',
    Field('firstname', 'string', length=50),
    Field('lastname', 'string', length=50))

db.define_table('author',
    Field('email', 'string', length=75, unique=True),
    Field('biography', 'text'),
    Field('photo', 'upload'),
    person)
```

Výpis 4: Ukázka mapování ve web2py DBAL

```
class Person(AbstractConcreteBase, Base):
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))

class Author(Person):
    __tablename__ = 'author'

    email = Column(String(75))
    biography = Column(Text)
    photo = String(String(100))

    __table_args__ = (UniqueConstraint('email'), )
    __mapper_args__ = {
        'polymorphic_identity': 'author',
        'concrete': True}
```

Výpis 5: Ukázka mapování v SQLAlchemy

4.4.2 Dotazování

```
>>> blog = Blog(name="New blog")
>>> blog.save()
#INSERT INTO "blogapp_blog" ("name", "description") VALUES ('New
    blog', '\') RETURNING "blogapp_blog"."id"
>>> blog.id
1
>>> blog.name = "New name"
>>> blog.save()
#UPDATE "blogapp_blog" SET "name" = 'New name', "description" =
    '\') WHERE "blogapp_blog"."id" = 1
>>> blog.entry_set.all()
#SELECT "blogapp_entry"."id", ... FROM "blogapp_entry" WHERE "
    blogapp_entry"."blog_id" = 1
>>> queryset = Entry.objects.filter(rating__gt=0)
>>> entry = queryset.order_by("headline")[:1][0]
#SELECT "blogapp_entry"."id", ... FROM "blogapp_entry" WHERE "
    blogapp_entry"."rating" > 0 ORDER BY "blogapp_entry"."headline"
    ASC LIMIT 1
>>> Blog.objects.raw('SELECT * FROM "blogapp_blog" ORDER BY "
    blogapp_blog"."name"')
#SELECT * FROM "blogapp_blog" ORDER BY "blogapp_blog"."name"
[<Blog: Blog object>]
```

Výpis 6: Ukázka práce s daty v Django ORM

```
>>> db.blog.insert(name="New blog")
#INSERT INTO blog(name) VALUES ('New blog');
1L
>>> db(db.blog.id == 1).update(name="New name")
#UPDATE blog SET name='New name' WHERE (blog.id = 1);
>>> blog = db(db.blog.id == 1).select()[0]
#SELECT blog.id, ... FROM blog WHERE (blog.id = 1);
>>> blog.entry.select()
#SELECT entry.id, ... FROM entry WHERE (entry.blog_id = 1);
>>> where = db.entry.rating > 0
>>> entry = db(where).select(orderby=db.entry.headline, limitby=(0
    ,1))[0]
#SELECT entry.id, ... FROM entry WHERE (entry.rating > 0) ORDER BY
    entry.headline LIMIT 1 OFFSET 0;
>>> db.executesql('SELECT * FROM blog ORDER BY blog.name')
#SELECT * FROM blog ORDER BY blog.name
[(1, 'New name', None)]
>>> db.commit()
```

Výpis 7: Ukázka práce s daty ve web2py DBAL

```
>>> Session = sessionmaker(bind=engine)
>>> session = Session()
>>> blog = Blog(name="New blog")
>>> session.commit()
#INSERT INTO blog (name, description) VALUES ('New name', NULL)
#RETURNING blog.id
>>> blog.id
#SELECT blog.id AS blog_id, ... FROM blog WHERE blog.id = 1
1
>>> blog.name = "New name"
>>> blog.entry.all()
#UPDATE blog SET name='New name' WHERE blog.id = 1
#SELECT entry.id AS entry_id, ... FROM entry WHERE 1= entry.
#blog_id
>>> query = session.query(Entry).filter(Entry.rating > 0)
>>> entry = query.order_by(Entry.headline)[:1][0]
#SELECT entry.id AS entry_id, ... FROM entry WHERE entry.raing > 0
#ORDER BY entry.headline LIMIT 1
>>> session.query(Blog).from_statement('SELECT * FROM blog ORDER
#BY blog.name')
#SELECT * FROM blog ORDER BY blog.name
[<blogapp.Blog at 0x9a950ac>]
>>> session.commit()
```

Výpis 8: Ukázka práce s daty v SQLAlchemy

5 Specifikace požadavků pro tvorbu nové technologie

Úkolem této diplomové práce bylo vytvořit vlastní knihovnu pracující na datové vrstvě aplikačního softwaru. Téma bylo původně diskutováno s firmou deving⁴. Tato firma se věnuje tvorbě informačních systémů (IS), systémů pro správu obsahu (CMS) a podnikových aplikací. Požadavkem firmy bylo vytvořit novou knihovnu nebo rámec na datové vrstvě, který by splňoval nároky na specifické použití pro tyto systémy.

Konkrétními požadavky firmy byly:

- **jazyk Python** Firma vytváří své aplikace pomocí programovacího jazyka Python. Pro zachování kompatibility a jednoduchosti je potřeba, aby datová vrstva byla implementována za použití tohoto jazyka.
- **rychlost získávání dat** Jedním z nejdůležitějších požadavků byl požadavek na celkový výkon datové vrstvy. Systémy vytvářené firmou jsou typické pro časté čtení a občasný zápis. Je tedy potřeba, aby vrstva urychlovala hlavně operace pro čtení.
- **konzistentní data, která drží relační schéma** Firma nesouhlasí s použitím objektové nebo dokumentově orientované databáze, jelikož výhody, které poskytují relační databázové systémy jsou pro vytváření systémů kritické. Datová vrstva by se tedy měla zaměřovat na databáze relační.
- **jednoduchost použití a čitelnost syntaxe** Datová vrstva by měla vyžadovat minimální znalosti pro její použití a měla by obsahovat pouze funkce, které se využijí při implementaci systémů. Syntaxe dotazovacího API by měla být čitelná a mělo by být jasné jaký dotaz se při spuštění provede. Syntaxe by měla být také jednodušší než samotné SQL nebo zápis SQL zjednodušovat.
- **stabilita** Výsledný produkt by měl být stabilní a měl by dodržovat základní principy čistého kódu. Struktura datové vrstvy a programového kódu by měla být přehledná a umožňovat jednoduché změny.
- **relační databáze PostgreSQL** Jako úložiště dat pro své systémy firma využívá SŘBD PostgreSQL. Firma SŘBD měnit neplánuje a nehodlá, a proto požaduje, aby se datová vrstva zaměřovala pouze na PostgreSQL.

Důvodem vytvoření nové knihovny byly neuspokojivé vlastnosti existujících rámců a knihoven. V sekci 4 byly popsány jedny z nejpoužívanějších existujících řešení pracujících na datové vrstvě pro jazyk Python. Žádné z těchto řešení nevyhovuje všem těmto požadavkům a jednotlivé nevýhody jsou popsány u každého řešení. Kromě těchto specifických nevýhod mají i další vlastnosti, které znesnadňují jejich použití v případech výše uvedených požadavků. Mezi tyto vlastnosti patří podpora několika SŘBD a O/R mapování.

⁴<http://deving.cz>

5.1 Objektově-relační mapování

Na základě požadavků jsem se rozhodl nevyužít funkcí objektově-relačního mapování. Požadavky odpovídají spíše datové vrstvě bez O/R mapování.

Pro O/R mapování existuje mnoho kritiky [17, 18, 19, 20]. Ve většině těchto případů se však jedná o špatné použití O/R mapování. O/R mapování není řešením pro všechny případy a snaha o jeho využití pro všechny situace vede k brzkému odhalení jeho limitů. ORM má své výhody a nevýhody. Pro mnoho projektů je ORM ideálním řešením, které ulehčí mnoho práce a času, pro některé případy však není vhodné.

Při použití O/R mapování je složité vytvořit jednoduché a čitelné dotazovací API do databáze tak, aby se operace, které pracují s databází neskrývaly za API ORM a bylo tak jasné, jaké dotazy se provádějí. Také je velmi pravděpodobné, že API nebude dostatečně abstrahovat databázi, což by znamenalo v případě složitějších struktur rozpad této abstrakce. Tento fakt je dán vzájemnou nekompatibilitou relačních a objektových struktur. Toto se tedy neslučuje s požadavkem na jednoduchost a čitelnost.

Databáze je pomocí ORM abstrahována, což většinou znamená, že všechny funkce databáze nejsou podporovány. Pokud je tomu tak, není využit maximální potenciál výkonu databáze, což se neslučuje s požadavkem na rychlost získávání dat.

Vytvořená knihovna tedy nebude implementovat O/R mapování.

5.2 Podpora různých SŘBD

Rámce nebo knihovny pracující na datové vrstvě často podporují mnoho různých SŘBD a abstrahují aplikační logiku od specifikací a detailů, aby tak umožnily SŘBD měnit podle potřeby. Aby bylo dosaženo nezávislosti aplikační logiky na konkrétní databázi, je možné abstrahovat pouze společnou množinu vlastností a funkcí všech podporovaných SŘBD. Dle požadavků je však vyžadována podpora pouze jediného SŘBD, což umožňuje zaměřit se více na tento konkrétní SŘBD, podporovat jeho specifické vlastnosti a optimalizovat datovou vrstvu tak, aby co nejvíce využívala konkrétního SŘBD a zvýšila tak celkový výkon systému.

Výsledná knihovna tedy bude implementována s myšlenkou, že bude vždy podporovat pouze jediný SŘBD. Tímto SŘBD je PostgreSQL.

5.2.1 Vlastnosti PostgreSQL

SŘBD PostgreSQL obsahuje mnoho specifických vlastností a funkcí, které nejsou mezi SŘBD velmi rozšířené a nejsou tak obsaženy v každém SŘBD. Některé z těchto vlastností by vytvořená knihovna mohla podporovat nebo jich využívat pro zlepšení rychlosti dotazování.

Rozmanité datové typy PostgreSQL umožňuje vytváření vlastních datových typů. Kromě toho již obsahuje různé datové typy, které umožňují ukládat například geografická data, JavaScriptový objektový zápis (JSON), XML nebo binární data.

Flexibilní datové typy Pomocí datového typu `hstore` umožňuje SŘBD ukládat data ve formě klíč-hodnota. Tento datový typ je bez schématu a při jeho použití lze získat variabilitu a flexibilitu dat podobně jako v NoSQL databázích. PostgreSQL navíc poskytuje pro tento datový typ speciální dotazovací syntaxi. Dále pak SŘBD obsahuje podporu i pro složené datové typy a objekty typu pole. Je tak možné do jednoho databázového sloupce uložit například všechny telefonní čísla pro konkrétního uživatele.

Funkce použité při dotazování Mezi nejzajímavější funkce, které lze použít při dotazování patří například funkce pro agregování jako `array_agg`, která vrací objekt typu pole, `bool_and`, který pro všechny prvky provede logický součin a nebo funkce `row_to_json`, která vrací každý řádek výsledku dotazu ve formátu JSON.

Podpora různých jazyků PostgreSQL umožňuje vytváření uživatelských funkcí pomocí procedurálních jazyků. Kromě standardního PL/pgSQL je možno využít i Python (PL/Python), Perl (PL/Perl) nebo i Ruby (PL/Ruby) [23].

Funkce nad okny Funkce nad okny (v angličtině Window Functions) fungují podobně jako v SŘBD Oracle. Pomocí těchto funkcí je možné provádět výpočty nad daty obdobně jako u agregačních funkcí, ovšem s tím rozdílem, že data není nutné seskupovat do skupin.

Dědičnost tabulek PostgreSQL podobně jako SŘBD Oracle dovoluje vytvářet tabulky s využitím dědičnosti. Tabulka tak získá všechny sloupce z děděné tabulky. Funguje zde i určitý druh polymorfismu, kdy na dotaz do bazové tabulky lze získat data i z odvozené tabulky.

Částečné indexy a omezení Při vytváření omezení je možné zadat podmínku, která bude určovat, kdy má dané omezení platit. Podobným způsobem lze vytvořit i indexy, což se hodí v případě, pokud není potřeba vytvářet index pro všechny hodnoty.

PostgreSQL obsahuje mnoho dalších funkcí a vlastností, jejich popis je však nad rámec této diplomové práce. Pro tento SŘBD navíc existuje i mnoho zásuvných modulů, které mu dodávají další funkce. Proč používat PostgreSQL a další jeho vlastnosti pěkně popisují dva články na internetu [21, 22].

5.3 Funkce pro zlepšení výkonu při čtení z databáze

Jedním z požadavků na vlastnosti vytvořené knihovny byla rychlá komunikace s databází, konkrétně rychlé načítání dat. Aby byl tento požadavek splněn, bude knihovna obsahovat funkce urychlující čtení dat. Mezi tyto funkce bude patřit simulace materializovaných pohledů a minimalizace přenášených dat.

5.3.1 Simulace materializovaných pohledů

SŘBD PostgreSQL obsahuje mnoho funkcí, ovšem materializované pohledy mezi nimi nejsou. Materializované pohledy jsou jednou z denormalizačních technik, které se použí-

vají pro urychlení čtení dat z databáze. Jsou podobné obyčejným databázovým pohledům, ale od těchto pohledů se liší realizací, která je řešená pomocí konstrukce podobné databázové tabulce. Tato tabulka obsahuje kopie dat získané pomocí SQL dotazu a slouží podobně jako keš pro data. Výhodou oproti obyčejným pohledům je možnost vytváření indexů nad touto tabulkou a rychlost čtení dat.

Pro zajištění konzistence dat je potřeba, aby v případě změny dat v tabulkách, které tvoří základ pro materializovaný pohled, byly aktualizovány i data v samotném materializovaném pohledu. To je možné buď ihned při změně dat nebo až při explicitním pokynutí.

Výsledná knihovna bude implementovat simulaci materializovaných pohledů, které budou aplikovat změny ihned při změně bazových dat. Materializovaný pohled bude simulován pomocí tabulky, která bude vytvořená na základě SQL dotazu, a pomocí triggerů, které budou aktualizovat materializovaný pohled v případě změny bazových tabulek.

5.3.2 Automatická denormalizace

Knihovna bude obsahovat mechanismus, který bude navrhopvat denormalizační změny. Na základě vstupních dat, které budou ve formě SQL dotazů a struktury databáze, mechanismus automaticky rozhodne, pro které struktury a jakou formou by bylo vhodné provést změny vedoucí k denormalizaci a ke zlepšení rychlosti čtení dat. Mechanismus se nebude soustředit pouze na zlepšení rychlosti čtení, ale bude brát v potaz i rychlost operací při aktualizaci dat, kterým případná denormalizace neprospěje. Výstupem této funkce budou zjištěné údaje, doporučené denormalizační změny a hodnoty časů ukazující, o kolik bude celkový výkon systému zlepšen.

Tato funkce tak zjednoduší uživateli rozhodování, na jakém místě by měl vytvořit výše popsaný simulovaný materializovaný pohled nebo jiné denormalizace.

5.3.3 Minimalizace přenášených dat

Knihovna se bude svými funkcemi snažit o co nejmenší objem přenášených dat. V případě pomalé odezvy databázového serveru je pro rychlost komunikace nesmírně kritické co nejméně přenesených dat pro získání výsledku dotazu.

Při pokládání dotazu pomocí spojení tabulek může nastat situace, kdy se přenáší stejná data vícekrát. V případě dlouhé odezvy nebo mnoha sloupců, lze data získat rychleji rozdělením dotazu na samostatné dotazy pro každou tabulku. Příklad zobrazuje tabulka 1. Tabulka je výsledkem dotazu z testovací databáze world⁵, který využívá spojení tabulek. Dotazem byla získána pro každé město i země, do které město náleží. Tabulka vypisuje pro přehlednost pouze část výsledku, ale i na tomto výsledku je však vidět, že každý záznam tabulky `country` se přenáší několikrát a vznikají několikanásobné kopie dat.

Tento problém popsal podrobněji Jakub Vrána na svém webu a řešení implementoval do své knihovny NotOrm [24]. V odkazovaném článku je proveden test, který dokazuje,

⁵<http://pgfoundry.org/projects/dbsamples/>

že je přístup samostatných dotazů v některých případech výhodnější než spojení tabulek. Test byl spouštěn na SRBD MySQL. Vyzkoušel jsem tedy podobný test pro PostgreSQL a výsledek zobrazuje tabulka 2.

city_id	city_name	country_name	country_continent	country_region
201	Sarajevo	Bosnia ...	Europe	Southern Europe
202	Banja Luka	Bosnia ...	Europe	Southern Europe
203	Zenica	Bosnia ...	Europe	Southern Europe
204	Gaborone	Botswana	Africa	Southern Africa
205	Francistown	Botswana	Africa	Southern Africa
206	São Paulo	Brazil	South America	South America
207	Rio de Janeiro	Brazil	South America	South America
208	Salvador	Brazil	South America	South America
209	Belo Horizonte	Brazil	South America	South America
210	Fortaleza	Brazil	South America	South America

Tabulka 1: Ukázka multiplicity dat při spojení tabulek

Podmínky	Spojení tabulek (s)	Samostatné dotazy (s)
Lokální server (PostgreSQL 9.2)	12.22	5.26
Vzdálený server (PostgreSQL 9.2)	54.34	38.71

Tabulka 2: Výsledky testování vlivu objemu přenášených dat na celkový výkon

V tomto testu byla použita opět databáze world a pro otestování byly položeny dotazy na získání prvních deseti měst, zemí, do kterých tyto města patří a jazyků, kterými se v těchto zemích mluví. Dotazy byly položeny pomocí spojení tabulek a pomocí samostatných dotazů. Tabulka obsahuje výsledné časy v sekundách pro tisíc spuštění a výsledek je nejmenším časem z pěti provedení.

Rozdíl je patrný i při použití poměrně jednoduchých testovacích dotazů. V praxi jsou používány daleko složitější konstrukce a rozdíly mezi těmito přístupy se tak projeví více.

Samostatné dotazy na každou tabulku tedy mohou v určitých případech urychlit získávání dat z databáze. Pro efektivnější provedení dotazu, jehož výsledek zobrazuje tabulka 1 se tedy provede nejdříve dotaz do databázové tabulky `city`, poté se uloží cizí klíče všech získaných záznamů odkazující na záznam v tabulce `country` a použijí se jako parametr podmínky dotazu do tabulky `country`. Výsledky obou dotazů se však dále musí před vypsáním finálního výsledku ještě sloučit podle cizího klíče. Konkrétní postup zobrazuje výpis 9.

Kód obstarávající tuto funkčnost je však monotónní a stává se komplikovaným při použití více tabulek a různých typů vazeb. Vytvořená knihovna tedy bude implementovat funkčnost, která tento monotónní kód bude provádět na pozadí a bude pro provádění samostatných dotazů poskytovat jednoduché API.

```
...
cursor.execute("SELECT id, countrycode, name FROM city")
cities = cursor.fetchall()
countrycodes = set([city['countrycode'] for city in cities])
cursor.execute("SELECT code, name, continent, region FROM country
    WHERE code IN ('%s')" % "','".join(countrycodes))
countries = cursor.fetchall()
countries_dict = {}
for country in countries:
    pk = country['code']
    countries_dict[pk] = country
for city in cities:
    fk = city['countrycode']
    country = countries_dict[fk]
    print city['name'], country['name'], country['continent'],
        country['region']
...
```

Výpis 9: Získávání dat pomocí samostatných dotazů

V některých případech však získávání dat pomocí samostatných dotazů není ideální. Samostatné dotazy mohou v určitých situacích z databáze přenášet zbytečné mezivýsledky, které by spojením tabulek byly vyfiltrovány. Vytvořená knihovna tedy kromě samostatných dotazů bude poskytovat možnost vytvořit dotazy do databáze i spojením tabulek.

6 Návrh nové technologie pracující na datové vrstvě

Vytvářenou knihovnu jsem nazval pypg. Knihovna bude splňovat všechny požadavky uvedené v sekci 5. Bude napsaná v jazyce Python a bude se specializovat na komunikaci se SŘBD PostgreSQL. Dále bude obsahovat jednoduché dotazovací API do databáze, které bude velmi podobné SQL. Knihovna bude malá, její architektura přehledná a požadavky na spuštění budou minimální. Bude obsahovat pouze funkce, které budou využívány při tvorbě systému firmou deving a svými funkcemi bude urychlovat operace pro čtení dat z databáze.

6.1 Klíčové vlastnosti

- Knihovna nebude obsahovat objektově-relační mapování a podobně jako web2py DBAL bude mapovat databázové tabulky a řádky na instance a třídy reprezentující množiny a záznamy. Vzhledem k tomu, že nebude použito O/R mapování se knihovna nebude řídit žádným návrhovým vzorem pro toto mapování. Mapování knihovny pypg však bude zčásti podobné návrhovému vzoru Table Data Gateway.
- Knihovna nebude obsahovat žádný mechanismus pro definování struktury v databázi. Vytvoření databázových tabulek knihovna nechá na uživateli. Při jejím používání tak nebude nutné tuto strukturu definovat a bude ji možné jednoduše použít s existující databází. Knihovna bude podporovat pouze jednoduché primární klíče.
- Rámce popsané v kapitole 4 používají k vytváření dotazů do databáze mapování struktur nadefinované uživatelem. Jelikož tuto funkci knihovna nebude obsahovat, bude metadata databázové struktury potřebné pro sestavování dotazů získávat pomocí introspekce. Kompletní schéma databáze tak bude zjištěno při startu knihovny z informačního schématu SŘBD PostgreSQL.
- Knihovna bude metadata databázového schématu používat i pro různá zjednodušení. Například budou tato data využita pro automatické kompletování při psaní kódu, pro výjimky knihovny nebo pro validování vytvořených dotazů. Zároveň však nebudou vyrazovány údaje o databázovém schématu, pokud bude knihovna nasazena v provozu.
- Introspekci dat nebude knihovna vyžadovat. Bude možné introspekci nahradit sadou předpisů, která bude určovat, jakými pravidly se řídí schéma v databázi. Toho bude možné dosáhnout rozšířením třídy `Naming` a předefinováním patřičných metod. Implicitně bude tato třída obsahovat pravidla, které budou určovat, že primární klíč každé tabulky má název `id` a každý cizí klíč odkazující se na tabulku `tabulka` má název `tabulka_id`.
- Knihovna nebude poskytovat žádnou validaci dat. Pro zajištění konzistence dat je lepší, pokud se data validují pouze v databázi, protože při přístupu do databáze přes jinou aplikaci může dojít k inkonzistenci dat [25].

6.2 Architektura

Při vytváření knihovny bude kladem zřetel na čistotu a jednoduchou strukturu kódu. Kniha Roberta Martina poslouží jako zdroj pravidel a postupů při vytváření knihovny pypg [26]. Celková funkčnost bude tak rozdělena do kompaktních jednoduchých tříd, které budou většinou v samostatných modulech. Názvy funkcí, metod a proměnných budou vytvořeny s rozvahou tak, aby nepotřebovaly komentáře. Komentáře v kódu zastarávají a často jim není věnována stejná pozornost při změnách, což vede ke zmateným informacím [26]. Jediným zdrojem pravdivých informací (SSOT) je kód. Knihovna tedy nebude obsahovat mnoho komentářů, ale bude napsána se snahou, aby kód popisoval sám sebe. Knihovna bude z 95% pokryta jednotkovými testy, které budou zjednodušovat případné změny⁶.

Níže jsou popsány třídy, které budou hlavní kostrou knihovny. Všechny tyto třídy zobrazuje diagram na obrázku 8.

PyPg Třída je vstupním bodem celé knihovny. Umožňuje nastavovat vlastnosti knihovny a poskytuje přístup k objektům třídy `Table`. Pomocí této třídy se spouští i vytváření materializovaných pohledů a automatická denormalizace.

Table Třída je rozhraním nad databázovou tabulkou. Pomocí svých metod umožňuje vytvářet dotazy do databáze. Poskytuje přístup k objektům třídy `Column`.

TableValidator Třída, která validuje dotazy před prováděním nad databází. Je využívána třídou `Table` a třídou `Row`.

RestrictedTableSelect Třída, která se používá při přístupu k relacím. Umožňuje vazební objekty filtrovat a poskytuje podobné metody jako třída `Table`.

ResultSet Reprezentace výsledku vráceného na SQL dotaz. Umožňuje funkci samostatných dotazů, která minimalizuje objem přenášených dat, a poskytuje přístup k instancím třídy `Row`.

Row Třída představuje řádek výsledku vráceného na SQL dotaz. Umožňuje s řádkem dále pracovat, poskytuje metody pro aktualizaci a mazání a metody pro přístup k vazebním objektům řádků.

ReadOnlyRow Reprezentace řádku z výsledku SQL dotazu, který používal spojení tabulek nebo neobsahuje primární klíče. Umožňuje data řádku pouze číst.

Column Třída představující databázový sloupec. Používá se při tvoření dotazů a poskytuje různé metody pro vyhledávání řádku. Vrací objekt třídy `Literal`.

Literal Reprezentace podmínky dotazu obsahující text a hodnoty. Třída `SQLBuilder` z instancí této třídy sestavuje SQL dotaz.

SQLBuilder Třída pro sestavování SQL dotazu.

⁶Pro zjištění pokrytí kódu testy bude využit Python coverage (<https://pypi.python.org/pypi/coverage>)

Query Třída, která přijímá hodnoty vrácené instancí třídy `SQLBuilder`, provádí SQL dotaz a získává výsledná data. Také shromažďuje statistiky pro automatickou de-normalizaci.

Manager Třída, která se stará o správu spojení a o nastavení knihovny.

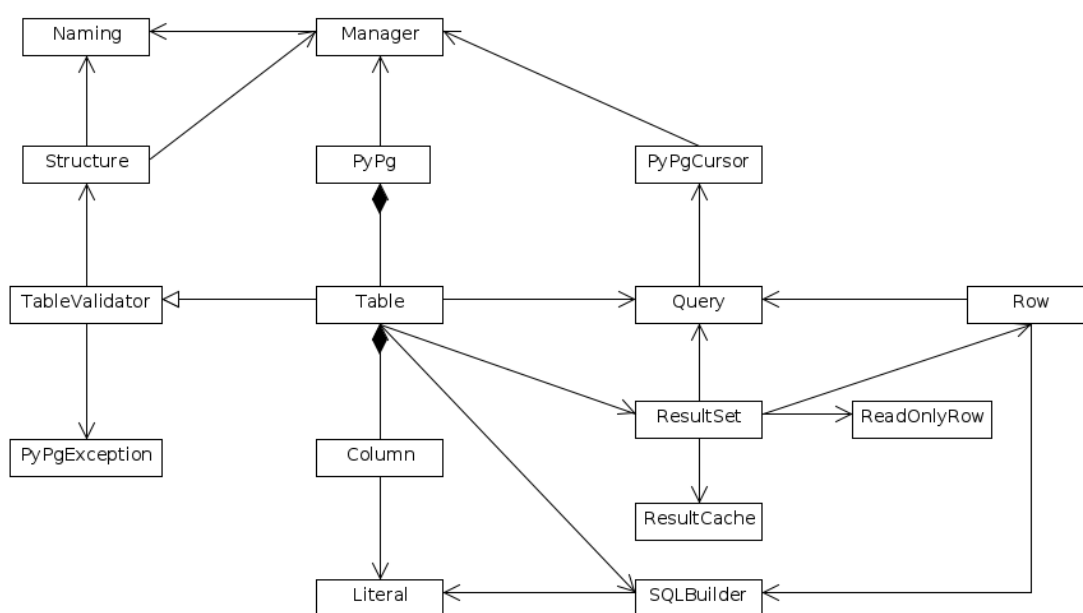
PyPgCursor Třída dědí ze třídy `psycopg2.extras.DictCursor` a pokud je to nastaveno, loguje dotazy pomocí standardního logování jazyka Python a pro každý dotaz měří čas provádění.

PyPgException Třída vytvořená pro výjimky vyvolané knihovnou.

Structure Třída, která abstrahuje strukturu databáze a poskytuje metody pro vyhledávání v této struktuře.

Naming Třída obsahuje pravidla, podle kterých se řídí struktura databáze. Pokud není povolena introspekce, je využívána třídou `Structure`.

ResultCache Třída uchovává data při použití funkce samostatných dotazů. Umožňuje v těchto datech vyhledávat a rozdělovat je podle příslušných vazeb. Je využívána třídou `ResultSet`.



Obrázek 8: Diagram tříd knihovny pypg

6.3 Funkce pro zrychlení čtení dat

Knihovna bude implementovat funkce, které urychlují a optimalizují čtení dat z databáze. Tyto funkce byly popsány v požadavcích v sekci 5.3.

6.3.1 Simulace materializovaných pohledů

Třída `MaterializedView` bude obsahovat logiku pro simulaci materializovaných pohledů. Tato třída bude umožňovat materializované pohledy vytvářet i mazat. Pro vytvoření materializovaného pohledu bude potřeba zavolat metodu `create_view` na instanci třídy `PyPg`. Metoda bude přijímat jako první parametr název materializovaného pohledu a jako druhý parametr dotaz vytvořený pomocí API knihovny.

Nejdříve bude dotaz validován a pomocí introspekce bude zjištěno, zda obsahuje primární klíče všech dotazovaných tabulek. Pokud ne, budou primární klíče do dotazu přidány, jelikož jsou potřeba při aktualizaci záznamů. Poté bude všem sloupcům v dotazu vytvořen alias složený z názvu tabulky a sloupce tak, aby se předešlo případným kolizím v názvech.

Pomocí výsledného dotazu bude v databázi vytvořen pohled a tento pohled bude pak použit jako zdroj dat při vytváření tabulky materializovaného pohledu. Pro tuto tabulku bude vytvořen primární klíč složený ze všech klíčů zjištěných dříve pomocí introspekce.

Pro každou tabulku obsaženou v dotazu budou vytvořeny trigger pro vložení, aktualizaci a smazání záznamu. Při aktualizaci záznamu budou v tabulce materializovaného pohledu podle primárního klíče vyhledány všechny korespondující záznamy. Tyto záznamy budou smazány a pak znovu načteny pomocí pohledu vytvořeného dříve. Aktualizace záznamů budou fungovat spolehlivě i při změně primárního klíče.

Celý proces zobrazuje diagram na obrázku 9.

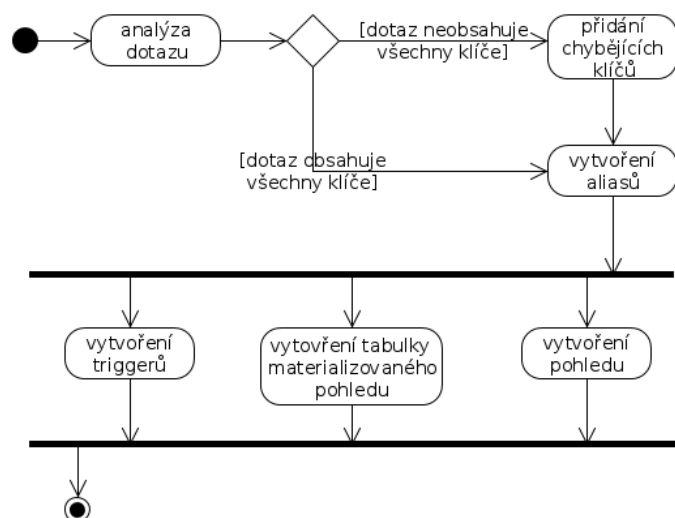
Metoda `drop_mview` třídy `PyPg` pak odstraní z databáze celou tuto logiku. Tabulku materializovaného pohledu, pohled pomocí, kterého byla tabulka vytvořena, a všechny trigger aktualizující data pro tento materializovaný pohled.

S vytvořeným materializovaným pohledem bude možné pracovat stejně jako s obyčejnou tabulkou. Podporovány budou pouze operace pro čtení dat, jelikož materializovaný pohled nebude obsahovat mechanismus, který by aktualizace dat aplikoval i na báze tabulky (více v sekci 6.5.3).

I když bude možné vytváření materializovaného pohledu spustit za běhu aplikace, mělo by být vytváření nebo mazání spouštěno mimo běh aplikace, jelikož je to časově náročná operace.

6.3.2 Automatická denormalizace

Celý proces automatické denormalizace bude započat při zapnutí logování statistik. Knihovna si vytvoří soubor, do kterého bude logovat všechny dotazy, které se budou provádět nad databází. Každý řádek v tomto souboru bude ve formátu JSON a bude obsahovat provedený dotaz, čas provádění dotazu a názvy sloupců a tabulek použitých v dotazu.



Obrázek 9: Diagram aktivit zobrazující vytváření materializovaného pohledu

Ve stejném okamžiku se provede i záloha současného stavu databáze, aby pak bylo možné simulovat stejné podmínky při testování denormalizace.

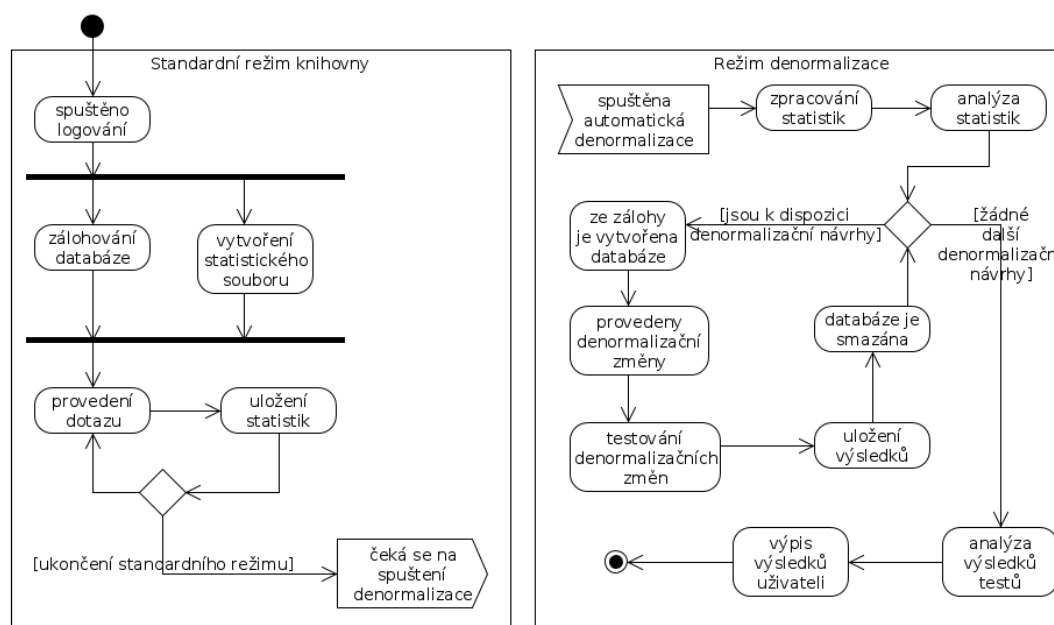
Poté, co bude knihovna nějaký čas v provozu a každý prováděný dotaz bude logován, bude možné spustit proces automatické denormalizace. Tento proces bude možné spustit pomocí metody `start_denormalization` třídy `PyPg` mimo běh aplikace. Před spuštěním bude doporučeno pomocí knihovny vytvořit spojení na testovací databázový server, jelikož produkční server by mohl být touto operací velmi zatěžován.

Třídy obstarávající funkčnost automatické denormalizace budou implementovány v modulu `pypg.denormalization`. Při spuštění procesu nejdříve pomocí třídy `Analyzer` proběhne analýza. Tato třída si načte data a statistiky uložené v souboru s logovanými dotazy. Tyto data poté zanalyzuje a zjistí případné denormalizační změny, které by mohly prospět rychlosti čtení dat z databáze. Pro denormalizační změny bude podmínka, že celkový čas dotazů strávený čtením nad konkrétní tabulkou musí být větší než čas strávený zápisem. Pokud tomu tak není, nebude mít smysl nad touto tabulkou provádět denormalizaci, jelikož to operace zápisu ještě více zpomalí. Třída poté vrátí kolekci všech doporučených denormalizačních změn v podobně instancí třídy `Improvement`. Tato kolekce bude obsahovat i seznam dotazů, které byly získány ze statistik, a které denormalizační změny ovlivní.

Každá denormalizační změna bude poté otestována nad testovací databází. Záloha databáze, získaná dříve, bude nahrána do testovací databáze. Denormalizace bude v databázi nejdříve vytvořena a poté budou spuštěny dotazy získané ze statistik, které interagují s tabulkami nebo strukturami ovlivněné denormalizací. Aby byly zaručeny stejné podmínky pro všechny testy, bude vždy před každým testem databáze převedena do stavu v jakém byla před testováním.

Výstupem bude seznam otestovaných denormalizačních změn, který bude obsahovat doporučení, zda konkrétní denormalizační změnu implementovat, a časy, které toto doporučení odůvodňují.

Celý proces automatické denormalizace lze vidět na obrázku 10. Mezi denormalizačními změnami bude podporován pouze materializovaný pohled.



Obrázek 10: Diagram aktivit zobrazující procesy automatické denormalizace

6.3.3 Minimalizace přenášených dat

Knihovna bude implementovat mechanismus popsáný v sekci 5.3.3. Tento mechanismus zmenšuje objem přenášených dat pomocí samostatných dotazů a tak i urychluje čtení z databáze. Celková funkčnost bude prováděna na pozadí a při dotazování nebude potřeba nic dalšího implementovat.

Jak bude funkce vypadat zobrazuje výpis 10. Tento výpis používá strukturu definovanou v sekci 4.4. Výstupem tohoto výpisu je seznam všech blogů a pro každý blog prvních deset příspěvků v tomto blogu a jejich autoři. Knihovna v tomto případě provede pouze 4 dotazy. Jeden dotaz pro získání dat z každé tabulky použité v tomto výpisu (`blog`, `entry`, `entry_authors` a `author`).

Knihovna bude rozpoznávat, že se výsledek dotazu prochází smyčkou, uloží si primární klíče všech záznamů, kterými je procházeno, a v případě přístupu k relaci za pomoci těchto klíčů vytvoří nový dotaz. Jelikož by nebylo efektivní vytvářet při každém přístupu k relaci nový dotaz (nastal by tak N+1 problém popsáný v sekci 4.1.3.2), vytvoří knihovna

takový dotaz, kterým budou získány všechny relace od všech záznamů najednou. Tyto relace potom bude distribuovat ke konkrétním záznamům.

```
>>>for blog in db.blog.select():
...     print 'Blog: ', blog['name']
...     print 'Příspěvky v blogu: '
...     for entry in blog.entry.limit(10).select():
...         print entry['headline']
...         print 'Autoři: '
...         for author_entry in entry.entry_authors.select():
...             author = author_entry.author_id
...             print author['firstname'], author['lastname']
```

Výpis 10: Návrh implementace samostatných dotazů v pypg

V případě druhého cyklu ve výpisu 10, kdy je dotazováno prvních 10 příspěvků blogu, bude tedy dotaz proveden pouze při prvním průchodu prvního cyklu a při dalších průchodech budou pak již využívána kešovaná data. Tento mechanismus bude fungovat nezávisle na počtu smyček.

Knihovna bude dotazy pro relační objekty skládat za pomoci introspekce a nebo pomocí pravidel struktury databáze. Seznam všech dotazů, které se provedou pro výpis 10, zobrazuje výpis 11.

```
SELECT * FROM "blog";
(SELECT * FROM "entry" WHERE entry.blog_id = 1LIMIT 10) UNION (SELECT * FROM "entry"
    WHERE entry.blog_id = 2LIMIT 10) UNION ...;
-- Do podmínky jsou použity hodnoty blog.id
SELECT * FROM "entry_authors" WHERE entry_id IN (1, 2, 3, 4...);
-- Do podmínky jsou použity hodnoty entry.id
SELECT * FROM "author" WHERE id IN (1, 2, 3, 4...);
-- Do podmínky jsou použity hodnoty entry_authors.author_id
```

Výpis 11: Dotazy vygenerované z výpisu 10 knihovnou pypg

6.4 Návrh syntaxe

Při návrhu syntaxe knihovny pypg byl kladen zřetel na jednoduchost a čitelnost dotazů, které se provádějí na pozadí. Zčásti byla syntaxe inspirována dotazovacím API web2py DBAL a SQLAlchemy a knihovnou NotORM. Podle mého názoru je tedy k jejímu porozumění potřeba minimum znalostí, pokud je uživatel již obeznámený se syntaxí SQL.

Ukázku navržené syntaxe zobrazuje výpis 12. Tento výpis používá schéma definované v sekci 4.4 a řídí se stejnými pravidly jako výpisy v sekci 4.4.1 tak, aby bylo možné tyto výpisy mezi sebou porovnat.

Všechny možnosti API knihovny pro práci s daty, sestavování dotazů a jejich kompletní popis lze nalézt v příloze D v uživatelské příručce.

```

>>> db = PyPg(conn, debug=True, strict=True)
>>> blog = db.blog.insert_and_get(name="New blog")[0]
#INSERT INTO "blog" ("name") VALUES ('New blog') RETURNING *
>>> blog['id']
1
>>> blog.update(name = "New name")
#UPDATE "blog" SET "name"='New name' WHERE "id"=1
>>> blog.entry.select()[0]
#SELECT * FROM "entry" WHERE "blog_id"=1
>>> where = db.entry.select().where(db.entry.rating > 0)
>>> entry = query.order(db.entry.headline).limit(1)[0]
#SELECT * FROM "entry" WHERE entry.rating > 0 ORDER BY entry.
    headline LIMIT 1
>>> Query().execute_and_fetch('SELECT * FROM "blog" ORDER BY blog.
    name')
#SELECT * FROM "blog" ORDER BY blog.name
[[1, 'New name', None]]

```

Výpis 12: Návrh syntaxe knihovny pypg

6.5 Funkce, které budou v budoucnu implementovány

Tyto funkce byly původně obsaženy v návrhu knihovny a počítalo se s jejich implementací. Postupem času se však ukázalo, že vzhledem k jejich časové náročnosti a nižší prioritě na základě požadavků popsaných v sekci 5, nebudou implementovány. Kvůli časovému omezení diplomové práce jsou tedy tyto funkce považovány nad rámec této práce.

Jak bylo popsáno výše, knihovna ovšem bude později používána jako prostředek pro tvorbu systémů firmy deving. Do budoucna se tedy plánuje s dalšími vylepšeními knihovny, mezi kterými budou i níže popsané funkce.

6.5.1 Tvoření dotazů pomocí SQL

Zadávání dotazů formou dotazovacího API knihovny je jednoduché, avšak můžou nastat případy, kdy toto API nebude vystačovat. V takovém případě by bylo dobré mít možnost, která umožní zadat dotaz ve formě SQL a přitom zachovat přístup ke všem funkcím knihovny.

Knihovna momentálně nepodporuje zadávání dotazů ve formě SQL. Je to dáno funkcemi v rámci knihovny, které pro svou funkčnost potřebují k dotazu získat i metadata. Například je u některých funkcí potřeba vědět, které tabulky a sloupce jsou v dotazu používány. Pokud jsou dotazy sestavovány pomocí API knihovny, lze tyto metadata jednoduše získat, jelikož jsou tvořeny při sestavování dotazu. Při použití SQL by však bylo potřeba nejdříve provést syntaktickou analýzu.

Knihovna tedy v budoucnu bude podporovat zadávání dotazů v podobě SQL a to jednoduchou formou, pro kterou lze provádět efektivní a rychlou syntaktickou analýzu. Všechny funkce, které knihovna obsahuje budou kompatibilní s touto formou dotazu.

Implementace této funkčnosti si bude brát inspiraci z knihovny dibi⁷.

6.5.2 Rozšíření možností dotazovacího API

API knihovny pypg pro generování dotazů momentálně umožňuje při sestavování dotazů používat pouze základní SQL výrazy. V budoucnu budou implementovány funkce, které budou umožňovat sestavovat SQL pomocí složitějších konstrukcí. Mezi tyto konstrukce budou patřit agregace, seskupování a specifické funkce SŘBD PostgreSQL.

6.5.3 Editovatelné pohledy

Knihovna momentálně podporuje pouze materializované pohledy. Tyto pohledy nelze využívat pro zápis dat, jelikož neexistuje mechanismus, který by propagoval změny do bazových tabulek. Vzhledem k požadavku na rychlost čtení dat popsaného v sekci 5, byla tato funkčnost přesunuta mezi budoucí rozšíření, jelikož by simulace editovatelného pohledu nijak nezvyšovaly rychlost čtení.

Tato funkce by však mohla ulehčit práci s daty materializovaného pohledu a bude tedy v budoucnu implementována.

6.5.4 Více možností denormalizace

Funkce automatické denormalizace momentálně navrhuje pouze zlepšení ve formě materializovaných pohledů. Materializovaný pohled však není jedinou formou denormalizace, a tak bude knihovna v budoucnu navrhopvat a testovat i jiné denormalizační techniky. Mezi podporované podoby denormalizace by mohlo patřit například přesunutí databázových sloupců na základě prováděných dotazů.

6.5.5 Minimalizace přenášených dat pomocí formátu JSON

V některých případech by při dotazování bylo vhodnější a efektivnější využít místo samostatných dotazů spojení tabulek. Jak už bylo zmíněno dříve v návrhu v sekci 5.3.3, spojení tabulek však přenáší více dat než samostatné dotazy.

V budoucnu bude tedy knihovna implementovat funkci, která bude využívat spojení tabulek, ale bude i minimalizovat data přenášená po síti. Toho bude dosaženo pomocí funkcí SŘBD PostgreSQL `row_to_json` a `array_agg` popsaných v sekci 5.2.1.

Z databáze budou data vrácena ve formátu JSON, který umožní jejich kompresi a zabrání přenášení několikanásobných kopií dat. Příklad zobrazuje výpis 13, kde jsou data z tabulky 1, která byla dříve několikanásobně přenášena, vrácena ve formě využívající formát JSON. Aby byl minimalizován objem dat, jsou atributům přiděleny aliasy, které zobrazuje první řádek v tomto výpisu. Tyto aliasy si knihovna uloží a nebudou přenášeny. Ve výpisu jsou pouze pro přehlednost.

⁷<http://dibiphp.com/>

```
{1: "country_name", 2: "country_continent", 3: "country_region",  
  4: "cities", 5: "city_id", 6: "city_name"}  
{1: "Bosnia and Herzegovina", 2: "Europe", 3: "Southern Europe",  
  4: [{5: 201, 6: "Sarajevo"}, {5: 202, 6: "Banja Luka"},  
    {5: 203, 6: "Zenica"}]}  
{1: "Botswana", 2: "Africa", 3: "Southern Africa",  
  4: [{5: 204, 6: "Gaborone"}, {5: 205, 6: "Francistown"}]}  
{1: "Brazil", 2: "South America", 3: "South America",  
  4: [{5: 206, 6: "Sao Paulo & Brazil"},  
    {5: 207, 6: "Rio de Janeiro"}, {5: 208, 6: "Salvador"},  
    {5: 209, 6: "Belo Horizonte"}, {5: 210, 6: "Fortaleza"}]}
```

Výpis 13: Minimalizace přenášných dat z tabulky 1 pomocí formátu JSON

Knihovna tato data poté na pozadí zpracuje a vypíše v požadované podobě.

Kromě této funkce bude knihovna na základě dotazu sestaveného pomocí API knihovny implicitně rozhodovat o jeho provedení pomocí samostatných dotazů nebo spojení tabulek s výsledkem vráceným ve formátu JSON.

7 Porovnání nové technologie s ostatními rámci

Rámce popsané v kapitole 4 byly spolu s knihovnou pypg podrobeny testům. Testy měly za úkol srovnat knihovnu s těmito rámci a ukázat výkon a efektivnost knihovny a výše popsaných funkcí, které knihovna implementuje.

7.1 Podmínky testování

Pro testování výkonu rámců byly vybrány podmínky, které by alespoň zčásti odpovídaly reálnému nasazení a požadavkům. Byla použita testovací databáze dellstore⁸, která představuje relační schéma jednoduchého internetového obchodu a obsahuje i náhodně vygenerovaná data. Tato databáze byla mírně upravena tak, aby dovolovala větší variabilitu testů. Do databáze byly přidány nové tabulky, které byly naplněny náhodně generovanými daty. Obrázek 11 zobrazuje relační schéma této databáze a tabulka 3, pak množství dat v jednotlivých databázových tabulkách. Vazební tabulky `product_category` a `product_variant` pak obsahují data tak, aby každý záznam v tabulce `product` obsahoval relaci na dva záznamy v tabulce `category` a čtyři záznamy v tabulce `variant`.

7.1.1 Použité technologie

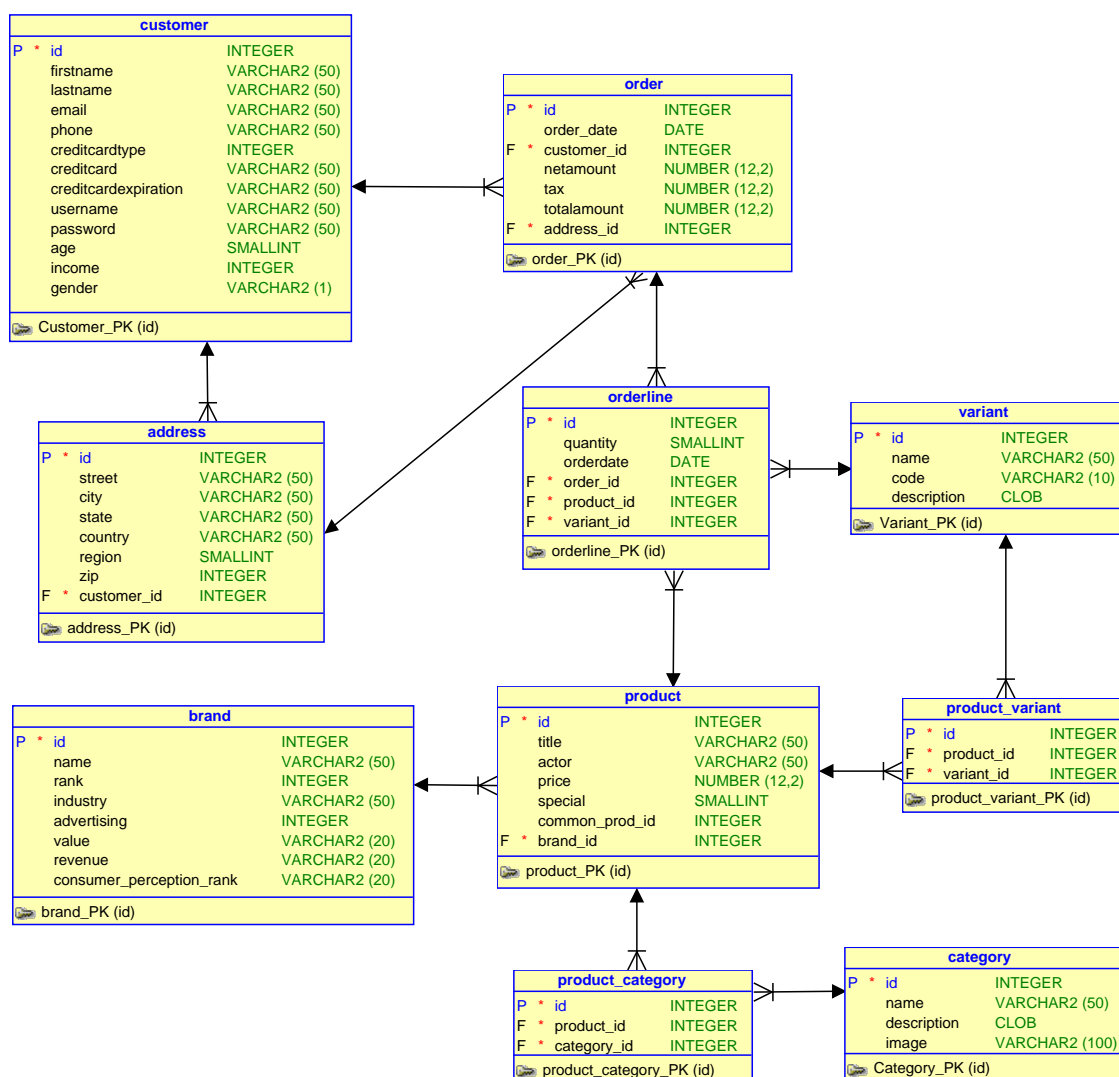
Testy byly spuštěny na zařízení s následujícím hardwarem:

- Procesor Intel Core i3-M330 s frekvencí 2.13GHz. Procesor obsahuje dvě jádra a čtyři vlákna.
- Pevný disk SATA HITACHI s obsahem 320 GB a s 5400 otáčky za minutu.
- Operační paměť 3 GB DDR3 s frekvencí 1066 MHz.

Software obsažený na tomto zařízení a jednotlivé verze použitých technologií a rámců byly tyto:

- Operační systém Ubuntu 12.04
- Lokální server SŘBD PostgreSQL 9.2.3
- Python 2.7.3
- Django 1.5
- web2py 2.4.6
- SQLAlchemy 0.8

⁸http://pgfoundry.org/frs/?group_id=1000150&release_id=376



Obrázek 11: ERD diagram testovací databáze

address	brand	category	customer	order	orderline	product	variant
30.000	100	16	20.000	12.000	60.350	10.000	512

Tabulka 3: Počet záznamů v tabulkách testovací databáze

7.2 Popis testů

V každém testu byly měřeny dva časové úseky. Prvním byla doba, kterou rámci nebo knihovně trvalo získat data z databáze. Do tohoto času spadá vyslání požadavku do databáze, zpracování požadavku databází a odeslání výsledných dat zpět rámci či knihovně. Druhým časovým úsekem bylo zpracování výsledných dat testovanou technologií do podoby, kdy je možné data vypsat na obrazovku nebo předat dále k dalšímu zpracování. První čas byl získán pomocí interních funkcí každého rámce nebo knihovny a pro získání druhého času byla použita knihovna jazyka Python `timeit`⁹. Do testů nebyly zahrnuty přípravné a ukončovací operace.

Před každým testem byla databáze vyprázdněna a znovu naplněna testovacími daty. Každý test byl spuštěn dvacetkrát a na výsledné časy byl aplikován medián.

Bylo vybráno šest testovacích případů, které představují typickou práci s daty v systémech, kde bude knihovna použita.

7.2.1 Test 1 - detail produktu

První test představuje detailní výpis jednoho záznamu z tabulky `product`. Spolu s každým produktem měly rámce a knihovna za úkol dále vypsat značku a varianty produktu a kategorie, do kterých produkt spadá.

Celkem tedy v tomto testu testované technologie získávají data z šesti tabulek v databázi - `product`, `brand`, `product_variant`, `variant`, `product_category`, `category`.

7.2.2 Test 2 - detail zákazníka

V tomto testu bylo za úkol vypsat detailní údaje jednoho zákazníka a všech jeho adres. Testované technologie získávaly data ze dvou tabulek v databázi - `customer` a `address`.

7.2.3 Test 3 - detail objednávky

Testované technologie v tomto testu měly za úkol vypsat detailní informace jedné objednávky. Pro každou objednávku byl vypsán zákazník, který objednávku vytvořil a adresa, která byla zadána do objednávky. Dále pak byly vypsány jednotlivé položky v objednávce, detaily produktů, jejich značky a kategorie, do kterých spadají a konkrétní varianty produktů. Celkem tedy byly získávány data z těchto tabulek v databázi: `order`, `customer`, `address`, `orderline`, `product`, `brand`, `product_category`, `category` a `variant`.

7.2.4 Test 4 - produkty v kategorii

Tento test sestával z detailního výpisu jedné kategorie a dvaceti produktů v této kategorii. Pro každý produkt byla pak dále vypsána jeho značka a jeho varianty.

Testované technologie tedy získávaly data z tabulky `category`, `product_category`, `product`, `brand`, `product_variant` a `variant`.

⁹<http://docs.python.org/2/library/timeit.html>

7.2.5 Test 5 - objednávky zákazníka

V tomto testu bylo za úkol vypsát detail zákazníka a všechny jeho objednávky. Maximálně však dvacet těchto objednávek. Dále pak pro každou objednávku všechny její položky. Celkem byla data získávána ze tří databázových tabulek - `customer`, `order` a `orderline`.

7.2.6 Test 6 - vytvoření nového zákazníka a objednávky

Všechny testy uvedené výše se zaměřují na rychlost čtení z databáze. Tento test však testoval rychlost operací při aktualizaci dat. V tomto testu měly testované technologie za úkol vytvořit nového zákazníka a jeho adresu. Pro tohoto zákazníka pak dále vytvořit objednávku se třemi položkami.

Byl tedy testován zápis do tabulek `customer`, `address`, `order` a `orderline`.

7.3 Vytváření testů

Testy byly vytvářeny tak, aby co nejvíce využívaly vlastností rámců a knihovny, a byly tak maximálně optimalizované pro danou technologii. Zároveň však nebyl vytvořen žádný vlastní kód, který by operace dále optimalizoval. Testy byly vytvořeny čistě pomocí API testovaných technologií.

Django ORM Při testování rámce byly využity optimalizace pro okamžité načtení vazebních objektů popsaných v sekci 4.1.3. Dále pak byla v testech využita funkce hromadného ukládání objektů.

web2py DBAL Testy pro tento rámec využívaly funkce hromadného ukládání objektů. Jiné optimalizační techniky, které by bylo možné využít v testech, rámec neposkytuje.

SQLAlchemy ORM Testy tohoto rámce využily funkcí pro optimalizované načítání vazebních objektů popsaných v sekci 4.3.3. Rámec některé další optimalizace poskytuje implicitně. Například hromadné ukládání nebo kešování objektů pomocí Identity Map v rámci transakce. Testy byly napsány s vědomím o těchto optimalizacích.

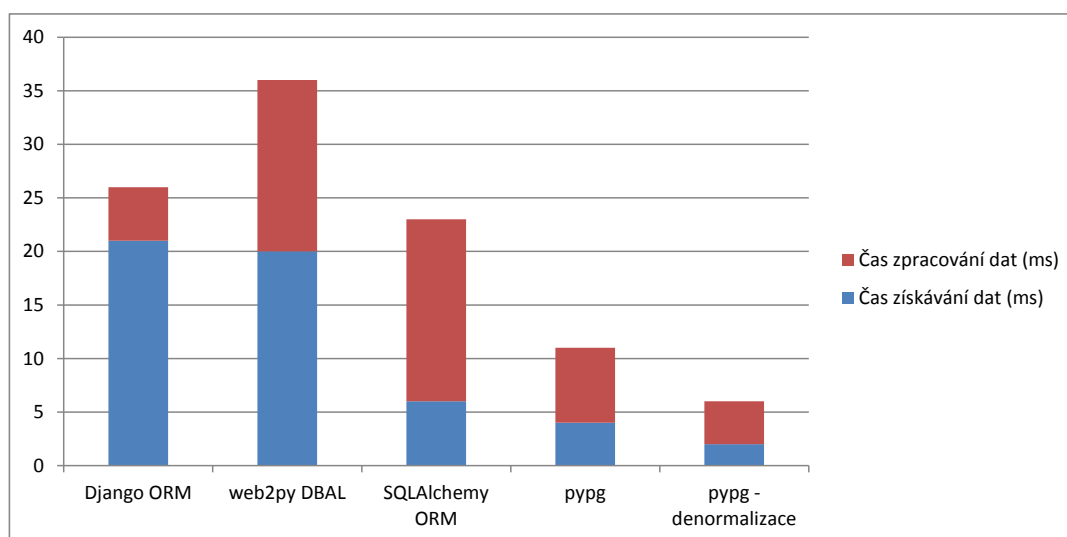
pypg Testy pro knihovnu pypg využívaly optimalizací popsaných v sekci 6.3. Testy byly napsány s využitím samostatných dotazů i dotazů vytvořených spojením tabulek. Na základě uložených statistik z těchto testů pak byla spuštěna automatická denormalizace. Celý výstup automatické denormalizace zobrazuje výpis 20 v příloze C.

Z tohoto výstupu jsem vybral dva knihovnou doporučené materializované pohledy a vytvořil je pomocí knihovny v testovací databázi. Testy pak byly vytvořeny tak, aby využívaly těchto optimalizačních změn.

7.4 Výsledky testů

7.4.1 Test 1 - detail produktu

Výsledky prvního testu zobrazuje graf na obrázku 12 a tabulka 4. Z výsledků je vidět, že knihovna pypg je několikanásobně lepší než ostatní rámce. Rozdíl se ještě více projeví, když knihovna používá navrhovaný materializovaný pohled. Naopak rámce Django ORM a web2py DBAL pokládaly neefektivní dotazy, což se projevilo na jejich celkovém čase.



Obrázek 12: Graf výsledků testu 1 - detail produktu

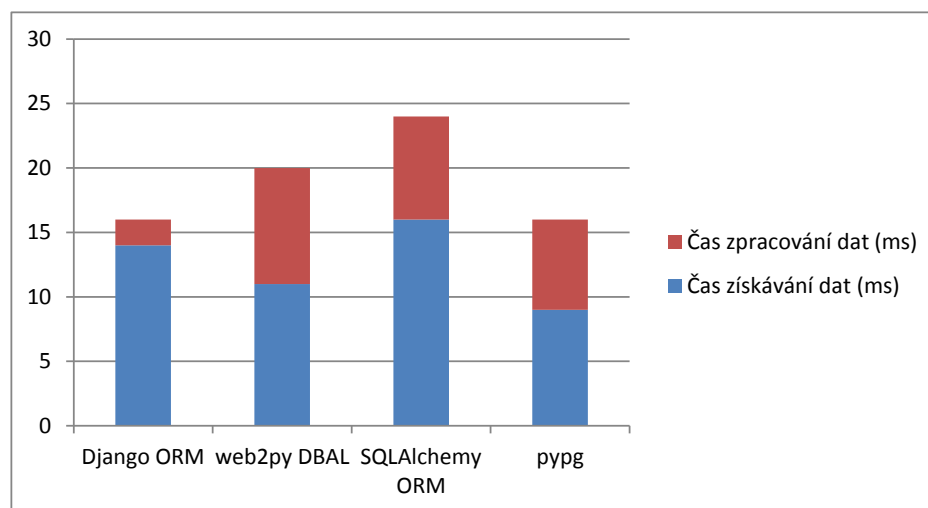
	Čas získání dat	Čas zpracování dat	Celkový čas
web2py DBAL	20 ms	16 ms	36 ms
Django ORM	21 ms	5 ms	26 ms
SQLAlchemy	6 ms	17 ms	23 ms
pypg	4 ms	7 ms	11 ms
pypg - po denormalizaci	2 ms	4 ms	6 ms

Tabulka 4: Tabulka výsledků testu 1 - detail produktu

7.4.2 Test 2 - detail zákazníka

Ve druhém testu nejsou patrné velké rozdíly mezi testovanými rámci a knihovnou. Je to dáno celkem jednoduchou strukturou testu. Knihovna pypg spolu s rámcem Django ORM dopadla nejlépe. Z dat lze vidět, že Django ORM dokázalo získaná data zpracovat

velmi rychle, což mu umožnilo splnit test nejrychleji. Materializované pohledy vytvořené knihovnou pypg neměly na tento test vliv. Výsledky testu zobrazuje graf na obrázku 13 a tabulka 5.



Obrázek 13: Graf výsledků testu 2 - detail zákazníka

	Čas získání dat	Čas zpracování dat	Celkový čas
SQLAlchemy	16 ms	8 ms	24 ms
web2py DBAL	11 ms	9 ms	20 ms
Django ORM	14 ms	2 ms	16 ms
pypg	9 ms	7 ms	16 ms
pypg - po denormalizaci	-	-	-

Tabulka 5: Tabulka výsledků testu 2 - detail zákazníka

7.4.3 Test 3 - detail objednávky

Výsledky třetího testu lze vidět na obrázku 14 a v tabulce 6. Knihovna pypg opět dopadla nejlépe. Navrhované denormalizační změny ovšem výkon moc nezlepšily, jelikož urychlily pouze část dotazů v testu. Materializované pohledy byly testovány funkcí automatické denormalizace knihovny pypg v celém kontextu dotazů. Celkově tyto pohledy urychlily získávání dat, ovšem v tomto testu zlepšení patrné nebylo.

Rámec web2py DBAL v tomto testu dopadl nejhůře. Jeho výsledky jsou dané zejména tím, že pomocí jeho API nebylo možné získat data spojením více než dvou tabulek nejspíše kvůli nějaké vnitřní chybě rámce. Jak bylo popsáno v sekci 4.2.4, stabilita rámce není

dokonalá. V testu tedy nastal u web2py DBAL N+1 problém, což se velmi negativně projevilo na výkonu.

	Čas získání dat	Čas zpracování dat	Celkový čas
web2py DBAL	78 ms	44 ms	122 ms
SQLAlchemy	14 ms	30 ms	44 ms
Django ORM	17 ms	16 ms	33 ms
pypg	7 ms	16 ms	23 ms
pypg - po denormalizaci	6 ms	14 ms	20 ms

Tabulka 6: Tabulka výsledků testu 3 - detail objednávky

7.4.4 Test 4 - produkty v kategorii

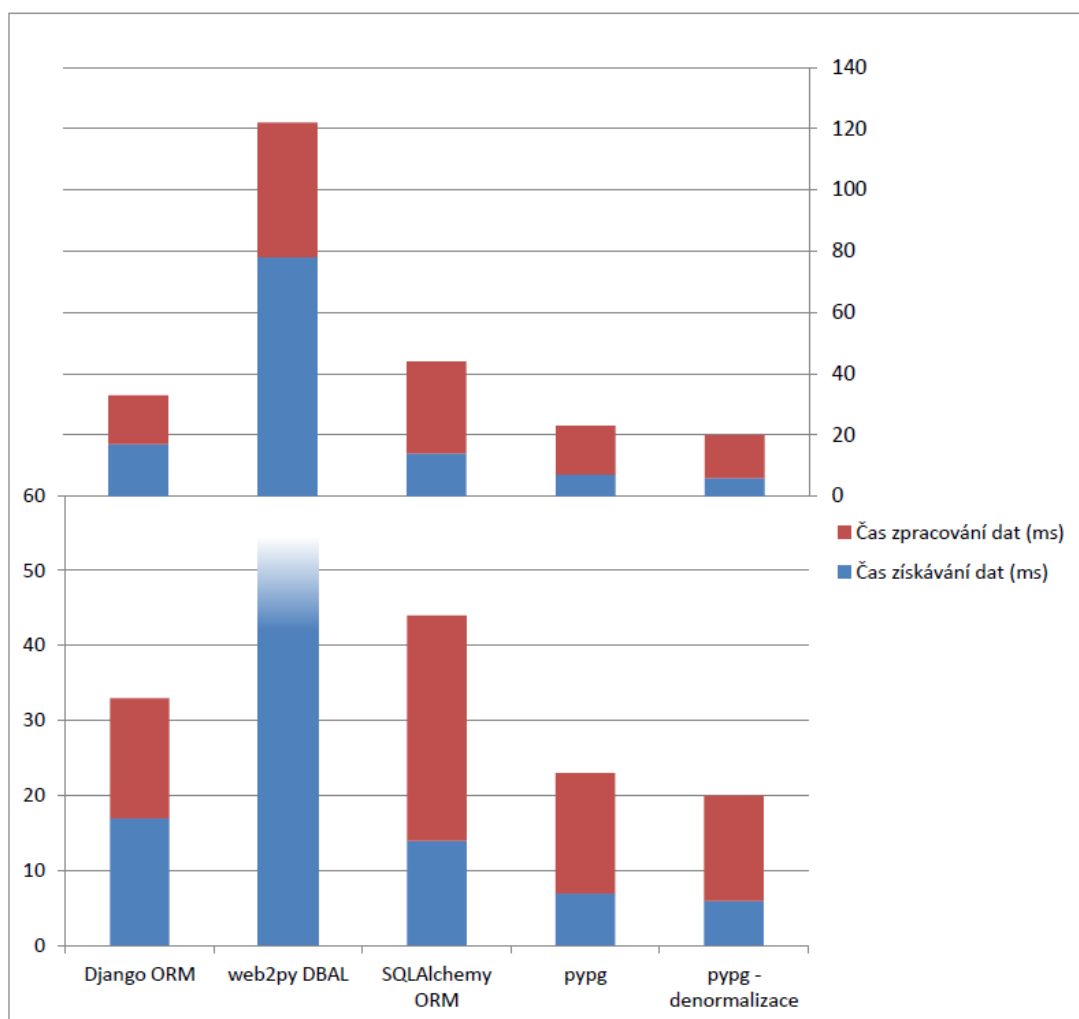
Ve čtvrtém testu nastal u rámce web2py DBAL stejný problém jako v testu třetím. Opět nebylo možné data pomocí API rámce získat spojením více než dvou tabulek a opět nastal N+1 problém. Tentokrát se ale následky projeví daleko hůře.

Knihovna pypg v tomto testu nedopadla nejlépe. I když položila velmi efektivní dotazy a byla schopná data získat několikanásobně rychleji než ostatní rámce, zpracování dat bylo velmi pomalé. Vytvořením materializovaného pohledu se zpracování dat mírně urychlilo a celkově se tak výsledky zlepšily.

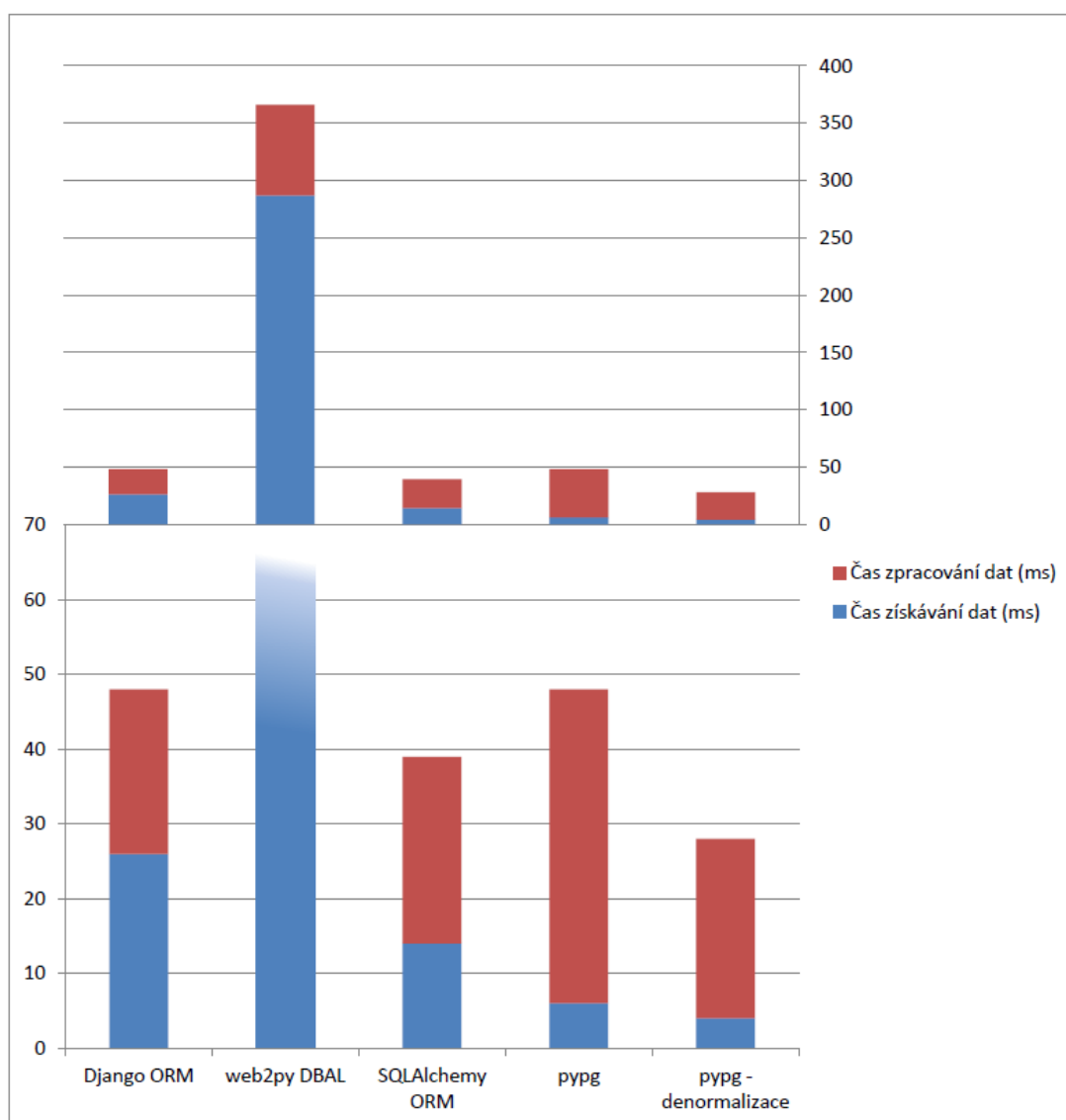
Tabulka 7 a graf na obrázku 15 zobrazují výsledky tohoto testu.

	Čas získání dat	Čas zpracování dat	Celkový čas
web2py DBAL	287 ms	79 ms	366 ms
Django ORM	26 ms	22 ms	48 ms
pypg	6 ms	42 ms	48 ms
SQLAlchemy	14 ms	25 ms	39 ms
pypg - po denormalizaci	4 ms	24 ms	28 ms

Tabulka 7: Tabulka výsledků testu 4 - produkty v kategorii



Obrázek 14: Graf výsledků testu 3 - detail objednávky

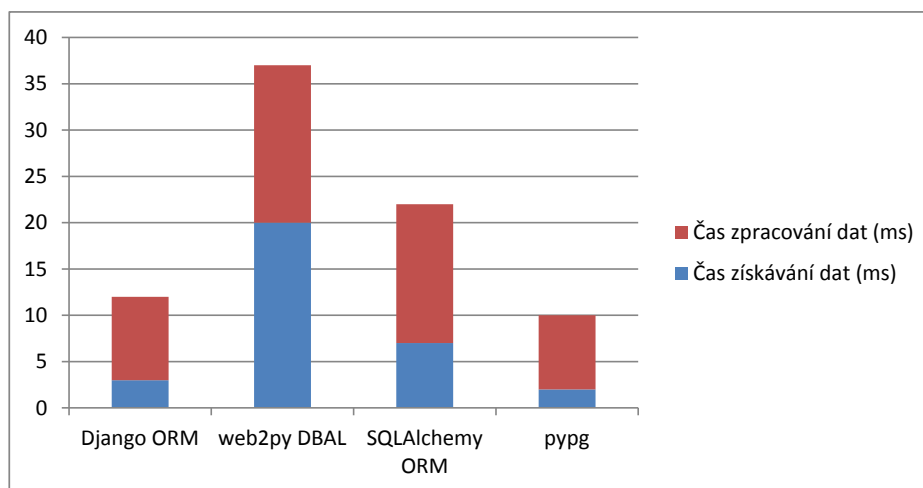


Obrázek 15: Graf výsledků testu 4 - produkty v kategorii

7.4.5 Test 5 - objednávky zákazníka

Výsledky pátého testu zobrazuje graf na obrázku 16 a tabulka 8. V tomto testu dopadl nejlépe rámec Django ORM spolu s knihovnou pypg. Rámec v tomto testu totiž použil podobné dotazy jako knihovna pypg a byl tak stejně efektivní. Ostatní rámce dopadly o dost hůře.

Materializované pohledy vytvořené knihovnou pypg na tento test neměly vliv.



Obrázek 16: Graf výsledků testu 5 - objednávky zákazníka

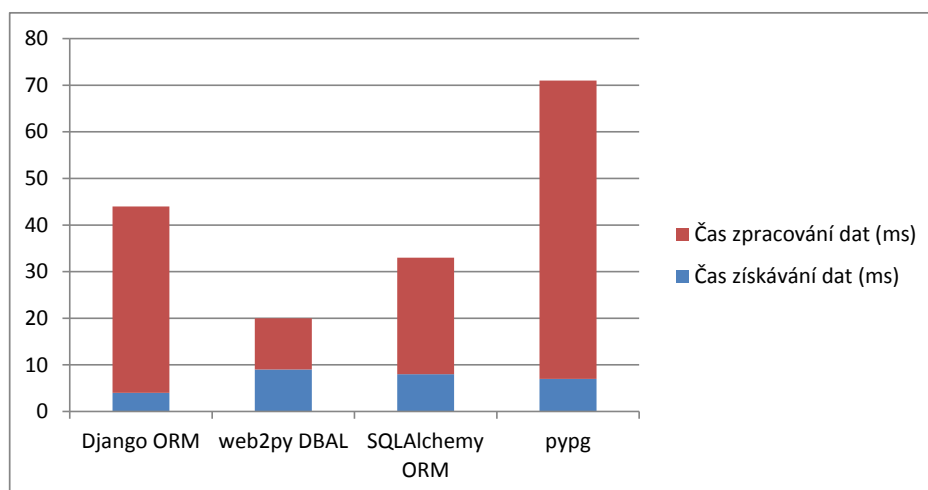
	Čas získání dat	Čas zpracování dat	Celkový čas
web2py DBAL	20 ms	17 ms	37 ms
SQLAlchemy	7 ms	15 ms	22 ms
Django ORM	3 ms	9 ms	12 ms
pypg	2 ms	8 ms	10 ms
pypg - po denormalizaci	-	-	-

Tabulka 8: Tabulka výsledků testu 5 - objednávky zákazníka

7.4.6 Test 6 - vytvoření nového zákazníka a objednávky

V šestém testu dopadla knihovna pypg nejhůře, jelikož neposkytuje žádné optimalizace pro zápis dat. Všechny ostatní rámce poskytují možnost hromadného ukládání. Rámec web2py dopadl nejlépe, jelikož nejefektivněji pracuje s ukládanými daty. Výsledky tohoto testu zobrazuje graf na obrázku 17 a tabulka 9.

Materializované pohledy vytvořené knihovnou pypg na tento dotaz neměly vliv.



Obrázek 17: Graf výsledků testu 6 - vytvoření nového zákazníka a objednávky

	Čas získání dat	Čas zpracování dat	Celkový čas
pypg	7 ms	54 ms	61 ms
Django ORM	4 ms	40 ms	44 ms
SQLAlchemy	8 ms	15 ms	33 ms
web2py DBAL	9 ms	11 ms	20 ms
pypg - po denormalizaci	-	-	-

Tabulka 9: Tabulka výsledků testu 6 - vytvoření nového zákazníka a objednávky

8 Závěr

Tato diplomová práce měla za úkol srovnat existující technologie pracující na datové vrstvě a vytvořit vlastní knihovnu, která by nad těmito technologiemi vynikala v rychlosti čtení dat z databáze. Všechny požadavky pro vytvoření knihovny byly splněny a knihovnu lze používat pro komunikaci s databází.

Diplomová práce ukázala, že vytváření nové technologie na datové vrstvě je sofistikovaná a časově náročná záležitost. Existující technologie, které byly popsány v této práci byly vyvíjeny několik let zkušenými programátory a jejich vývoj není zdaleka u konce. Proto vidím jako úspěch, že se mnou vytvořená knihovna dokázala těmto technologiím vyrovnat a dokonce je dokázala předčit v rychlosti čtení dat z databáze. Studium jednotlivých technologií, jejich funkcí a vlastností jsem dokázal zjistit jejich slabiny a silné stránky, což mi umožnilo vytvořit knihovnu, která by tyto technologie v rychlosti čtení dat předčila.

Testy v poslední kapitole však ukázaly, že při čtení dat z databáze nezáleží z velké části pouze na optimalizaci SQL dotazů a databázové struktury, ale i velmi na zpracování těchto dat datovou vrstvou. I když byla knihovna vytvářena tak, aby neobsahovala náročné operace, které by zpracovávání dat brzdily, myslím, že byl tento fakt mírně podceněn, což se projevilo v jednom z výše uvedených testů, a další vývoj knihovny by se tedy měl zaměřovat více i na optimalizaci struktur knihovny samotné.

I když knihovna nad existujícími technologiemi vyniká v rychlosti čtení dat z databáze, v některých oblastech oproti těmto technologiím zaostává. Při vývoji jsem kvůli časovému omezení diplomové práce musel vybírat, které vlastnosti bude výsledná knihovna obsahovat. Rozhodoval jsem se na základě požadavků a zčásti i na faktu, aby mnou vytvořená technologie alespoň v jednom bodě předčila technologie existující, což se mi povedlo.

Vývoj knihovny bude dále pokračovat, jelikož bude využívána při tvorbě systému firmy deving. Knihovna bude rozšířena o funkce popsané v této práci i o další funkce, kterými jsem byl inspirován při studiu existujících technologií.

Tato diplomová práce je podobná svou strukturou předchozím diplomovým pracím, jejichž výstupem byly rámce BasicORM [29] a EasyORM [30]. Od těchto prací se má diplomová práce lišit jiným přístupem k realizaci technologie na datové vrstvě. Oba rámce využívají objektově-relačního mapování. EasyORM je vytvořen pro platformu Java a BasicORM pro platformu .NET. Knihovna pypg byla vytvořena bez objektově-relačního mapování pomocí programovacího jazyka Python a zaměřuje se více na výkon a rychlost čtení dat z databáze.

9 Reference

- [1] FOWLER, Martin. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2003. ISBN 0321127426.
- [2] Chapter 5: Layered Application Guidelines. In: MICROSOFT PATTERNS & PRACTICES TEAM. Microsoft Application Architecture Guide, 2nd Edition [online]. 2009 [cit. 2013-04-24]. Dostupné z: <http://msdn.microsoft.com/en-us/library/ee658109.aspx>
- [3] FOWLER, Martin. InversionOfControl. In: Martin Fowler [online]. 2005, 26.6. [cit. 2013-04-24]. Dostupné z: <http://martinfowler.com/bliki/InversionOfControl.html>
- [4] DOWNS, Kenneth. Historical Perspective of ORM and Alternatives. In: The Database Programmer [online]. 2010, 15.12. [cit. 2013-04-24]. Dostupné z: <http://database-programmer.blogspot.cz/2010/12/historical-perspective-of-orm-and.html>
- [5] DRESLER, Robert. Vícevrstvé architektury aplikací. In: Robert Dresler: Vývoj software není jenom o kódování ... [online]. 2011, 26.4. [cit. 2013-04-24]. Dostupné z: <http://www.robertdresler.cz/2011/04/vicevrstve-architektury-aplikaci.html>
- [6] Web Frameworks for Python. Python Programming Language – Official Website [online]. 2013, 23.4. [cit. 2013-04-24]. Dostupné z: <http://wiki.python.org/moin/WebFrameworks>
- [7] Framework Django v kostce. Django Česká republika [online]. 2005 [cit. 2013-04-24]. Dostupné z: <http://www.djangoproject.cz/>
- [8] Adrian Holovaty. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001-2013, 8.2.2013 [cit. 2013-04-24]. Dostupné z: http://en.wikipedia.org/wiki/Adrian_Holovaty
- [9] Django documentation. DJANGO SOFTWARE FOUNDATION. Django the Web framework for perfectionists with deadlines [online]. 2005-2013 [cit. 2013-04-24]. Dostupné z: <https://docs.djangoproject.com/en/1.5/>
- [10] Making queries. In: DJANGO SOFTWARE FOUNDATION. Django documentation [online]. 2005-2013 [cit. 2013-04-24]. Dostupné z: <https://docs.djangoproject.com/en/1.5/topics/db/queries/>
- [11] QuerySet API reference. In: DJANGO SOFTWARE FOUNDATION. Django documentation [online]. 2005-2013 [cit. 2013-04-24]. Dostupné z: <https://docs.djangoproject.com/en/1.5/ref/models/querysets/>
- [12] Model instance reference. In: DJANGO SOFTWARE FOUNDATION. Django documentation [online]. 2005-2013 [cit. 2013-04-24]. Dostupné z: <https://docs.djangoproject.com/en/1.5/ref/models/instances/>

-
- [13] Why I hate the Django ORM. In: GAYNOR, Alex. YouTube [online]. 2012 [cit. 2013-04-24]. Dostupné z: <http://www.youtube.com/watch?v=GxL9MnWlCwo>
- [14] Web2py. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001-2013 [cit. 2013-04-24]. Dostupné z: <http://en.wikipedia.org/wiki/Web2py>
- [15] DI PIERRO, Massimo. What is web2py. Web2py Web Framework [online]. 2013 [cit. 2013-04-24]. Dostupné z: <http://web2py.com/init/default/what>
- [16] The database abstraction layer. In: DI PIERRO, Massimo. Web2py Complete Reference Manual [online]. 5th Edition. 2007-2013 [cit. 2013-04-24]. Dostupné z: <http://web2py.com/books/default/chapter/29/06>
- [17] VOSS, Laurie. ORM is an anti-pattern. In: Seldo.com [online]. 2011, 11.8. [cit. 2013-04-24]. Dostupné z: <http://seldo.com/weblog/2011/08/11/orm.is.an.antipattern>
- [18] FOWLER, Martin. OrmHate. In: Martin Fowler [online]. 2012, 8.5. [cit. 2013-04-24]. Dostupné z: <http://martinfowler.com/bliki/OrmHate.html>
- [19] DOWNS, Kenneth. Why I Do Not Use ORM. In: The Database Programmer [online]. 2008, 15.6. [cit. 2013-04-24]. Dostupné z: <http://database-programmer.blogspot.cz/2008/06/why-i-do-not-use-orm.html>
- [20] ORM anti-pattern series. KHALILI, Mehdi Khalili [online]. 2011, 12.2. [cit. 2013-04-24]. Dostupné z: <http://www.mehdi-khalili.com/orm-anti-patterns-series>
- [21] KERSTIENS, Craig. Why Postgres. In: Craig Kerstiens [online]. 2012, 30.4. [cit. 2013-04-24]. Dostupné z: <http://craigkerstiens.com/2012/04/30/why-postgres/>
- [22] KERSTIENS, Craig. Why PostgreSQL Part 2. In: Craig Kerstiens [online]. 2012, 7.5. [cit. 2013-04-24]. Dostupné z: <http://craigkerstiens.com/2012/05/07/why-postgres-part-2/>
- [23] Chapter 38. Procedural Languages. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL Documentation [online]. 2013 [cit. 2013-04-24]. Dostupné z: <http://www.postgresql.org/docs/9.2/static/xplang.html>
- [24] VRÁNA, Jakub. Srovnání dotazů do závislých tabulek. In: PHP triky [online]. 2009, 27.2. [cit. 2013-04-24]. Dostupné z: <http://php.vrana.cz/srovnani-dotazu-do-zavislych-tabulek.php>
- [25] Proč NotORM. VRÁNA, Jakub. NotORM - PHP library for simple working with data in the database [online]. 2010 [cit. 2013-04-24]. Dostupné z: <http://www.notorm.com/#why>
- [26] MARTIN, Robert C. Clean code : a handbook of agile software craftsmanship. NJ : Prentice Hall: Upper Saddle River, 2009. ISBN 0132350882.

- [27] SQLAlchemy. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001-2013 [cit. 2013-05-04]. Dostupné z: <http://en.wikipedia.org/wiki/SQLAlchemy>
- [28] BAYER, Mike. SQLAlchemy Documentation. Release 0.8.0b2. 7. 3. 2013. Dostupné z: http://media.readthedocs.org/pdf/sqlalchemy/rel_0.8/sqlalchemy.pdf
- [29] JEŘÁBEK, Tomáš. Implementace objektově-relačního mapování v datové vrstvě informačního systému pro platformu .NET. Ostrava, 2011. Diplomová práce. Vysoká škola báňská.
- [30] TURČÍK, Lubomír. Implementace objektově-relačního mapování v datové vrstvě informačního systému pro platformu Java. Ostrava, 2010. Diplomová práce. Vysoká škola báňská.

A Výpisy mapování rámců ze sekce 4.4

```
from django.db import models

class Person(models.Model):
    firstname = models.CharField(max_length=50)
    lastname = models.CharField(max_length=50)

    class Meta:
        abstract = True

class Author(Person):
    email = models.EmailField(unique=True)
    biography = models.TextField()
    photo = models.ImageField(upload_to="photos/")

class Blog(models.Model):
    name = models.CharField(max_length=100, db_index=True)
    description = models.TextField()

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField(auto_now_add=True)
    mod_date = models.DateField(auto_now=True)
    authors = models.ManyToManyField(Author)
    comments = models.IntegerField()
    rating = models.IntegerField()

class VlogEntry(Entry):
    video = models.URLField()
```

Výpis 14: Schéma jednoduché aplikace pro vytváření blogů v Django ORM

```
db = DAL("postgres://username:password@localhost/database")

person = db.Table(db, 'person',
    Field('firstname', 'string', length=50),
    Field('lastname', 'string', length=50))

db.define_table('author',
    Field('email', 'string', length=75, unique=True),
    Field('biography', 'text'),
    Field('photo', 'upload'),
    person)

db.define_table('blog',
    Field('name', 'string', length=100),
    Field('description', 'text'))

now = lambda: datetime.datetime.now()
db.define_table('entry',
    Field('blog_id', db.blog),
    Field('headline', 'string'),
    Field('bodytext', 'text'),
    Field('pub_date', 'date', default=now),
    Field('mod_date', 'date', default=now, update=now),
    Field('bodytext', 'text'),
    Field('comments', 'integer'),
    Field('rating', 'integer'))

db.define_table('vlogentry', db.entry, Field('video', 'string'))

db.define_table('entry_author',
    Field('entry', db.entry),
    Field('author', db.author))
```

Výpis 15: Schéma jednoduché aplikace pro vytváření blogů ve web2py DBAL

```

from sqlalchemy.ext.declarative import declarative_base,
    AbstractConcreteBase
from sqlalchemy import Column, Integer, String, Text, Date,
    ForeignKey, Table, UniqueConstraint, Index
from sqlalchemy.orm import relationship, backref
import datetime

Base = declarative_base()

entry_authors = Table('entry_authors', Base.metadata,
    Column(u'entry_id', Integer, ForeignKey('entry.id')),
    Column(u'author_id', Integer, ForeignKey('author.id')),
)
class Person(AbstractConcreteBase, Base):
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))

class Author(Person):
    __tablename__ = 'author'

    email = Column(String(75))
    biography = Column(Text)
    photo = Column(String(100))

    __table_args__ = (UniqueConstraint('email'), )
    __mapper_args__ = {
        'polymorphic_identity': 'author',
        'concrete': True}

class Blog(Base):
    __tablename__ = 'blog'

    id = Column(Integer, primary_key=True)
    name = Column(String(100))
    description = Column(Text)

    __table_args__ = (Index('blog_name_idx', "name"), )

class Entry(Base):
    __tablename__ = 'entry'

    id = Column(Integer, primary_key=True)
    headline = Column(String(255))
    body_text = Column(Text)
    pub_date = Column(Date, default=datetime.datetime.now())
    mod_date = Column(Date, onupdate=datetime.datetime.now())
    comments = Column(Integer)

```

```
rating = Column(Integer)
blog_id = Column(Integer, ForeignKey('blog.id'))
type = Column(String(50))

blog = relationship("Blog", backref=backref('entries', lazy='
dynamic'))
authors = relationship("Author", secondary=entry_authors,
backref=backref('entries', lazy='dynamic'))

__mapper_args__ = {
    'polymorphic_identity': 'entry',
    'polymorphic_on': type
}

class VlogEntry(Base):
    __tablename__ = 'vlogentry'

    entry_id = Column(Integer, ForeignKey('entry.id'), primary_key=
True)
    video = Column(String(100))

    __mapper_args__ = {
        'polymorphic_identity': 'vlogentry'
    }
```

Výpis 16: Schéma jednoduché aplikace pro vytváření blogů v SQLAlchemy

B Výpisy SQL vygenerované rámci ze sekce 4.4

```

CREATE TABLE "blogapp_blog" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(100) NOT NULL,
    "description" text NOT NULL
)
;
CREATE TABLE "blogapp_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(50) NOT NULL,
    "last_name" varchar(50) NOT NULL,
    "email" varchar(75) NOT NULL UNIQUE,
    "biography" text NOT NULL,
    "photo" varchar(100) NOT NULL
)
;
CREATE TABLE "blogapp_entry_authors" (
    "id" serial NOT NULL PRIMARY KEY,
    "entry_id" integer NOT NULL,
    "author_id" integer NOT NULL REFERENCES "blogapp_author" ("id") DEFERRABLE
        INITIALLY DEFERRED,
    UNIQUE ("entry_id", "author_id")
)
;
CREATE TABLE "blogapp_entry" (
    "id" serial NOT NULL PRIMARY KEY,
    "blog_id" integer NOT NULL REFERENCES "blogapp_blog" ("id") DEFERRABLE INITIALLY
        DEFERRED,
    "headline" varchar(255) NOT NULL,
    "body_text" text NOT NULL,
    "pub_date" date NOT NULL,
    "mod_date" date NOT NULL,
    "comments" integer NOT NULL,
    "rating" integer NOT NULL
)
;
ALTER TABLE "blogapp_entry_authors" ADD CONSTRAINT "entry_id_refs_id_162c5c54" FOREIGN
    KEY ("entry_id") REFERENCES "blogapp_entry" ("id") DEFERRABLE INITIALLY DEFERRED;
CREATE TABLE "blogapp_vlogentry" (
    "entry_ptr_id" integer NOT NULL PRIMARY KEY REFERENCES "blogapp_entry" ("id")
        DEFERRABLE INITIALLY DEFERRED,
    "video" varchar(200) NOT NULL
)
;
CREATE INDEX "blogapp_blog_name" ON "blogapp_blog" ("name");
CREATE INDEX "blogapp_blog_name_like" ON "blogapp_blog" ("name" varchar_pattern_ops);
CREATE INDEX "blogapp_entry_blog_id" ON "blogapp_entry" ("blog_id");

```

Výpis 17: Reprezentace modelů z výpisu 14 v PostgreSQL vygenerovaná rámcem Django ORM

```
CREATE TABLE author(  
  id SERIAL PRIMARY KEY,  
  email VARCHAR(512) UNIQUE,  
  biography TEXT,  
  photo VARCHAR(512),  
  first_name VARCHAR(50)  
);  
CREATE TABLE blog(  
  id SERIAL PRIMARY KEY,  
  name VARCHAR(100),  
  description TEXT  
);  
CREATE TABLE entry(  
  id SERIAL PRIMARY KEY,  
  blog_id INTEGER REFERENCES blog (id) ON DELETE CASCADE,  
  headline VARCHAR(512),  
  bodytext TEXT,  
  pub_date DATE,  
  mod_date DATE,  
  comments INTEGER,  
  rating INTEGER  
);  
CREATE TABLE vlogentry(  
  id SERIAL PRIMARY KEY,  
  blog_id INTEGER REFERENCES blog (id) ON DELETE CASCADE,  
  headline VARCHAR(512),  
  bodytext TEXT,  
  pub_date DATE,  
  mod_date DATE,  
  comments INTEGER,  
  rating INTEGER,  
  video VARCHAR(512)  
);  
CREATE TABLE entry_author(  
  id SERIAL PRIMARY KEY,  
  entry INTEGER REFERENCES entry (id) ON DELETE CASCADE,  
  author INTEGER REFERENCES author (id) ON DELETE CASCADE  
);
```

Výpis 18: Reprezentace schéma z výpisu 15 v PostgreSQL vygenerovaná rámcem web2py DBAL

```
CREATE TABLE blog (  
    id SERIAL NOT NULL,  
    name VARCHAR(100),  
    description TEXT,  
    PRIMARY KEY (id)  
)  
COMMIT  
CREATE INDEX blog_name_idx ON blog (name)  
COMMIT  
CREATE TABLE author (  
    id SERIAL NOT NULL,  
    firstname VARCHAR(50),  
    lastname VARCHAR(50),  
    email VARCHAR(75),  
    biography TEXT,  
    PRIMARY KEY (id),  
    UNIQUE (email)  
)  
COMMIT  
CREATE TABLE entry (  
    id SERIAL NOT NULL,  
    headline VARCHAR(255),  
    body_text TEXT,  
    pub_date DATE,  
    mod_date DATE,  
    comments INTEGER,  
    rating INTEGER,  
    blog_id INTEGER,  
    type VARCHAR(50),  
    PRIMARY KEY (id),  
    FOREIGN KEY(blog_id) REFERENCES blog (id)  
)  
COMMIT  
CREATE TABLE entry_authors (  
    entry_id INTEGER,  
    author_id INTEGER,  
    FOREIGN KEY(entry_id) REFERENCES entry (id),  
    FOREIGN KEY(author_id) REFERENCES author (id)  
)  
COMMIT  
CREATE TABLE vlogentry (  
    entry_id INTEGER NOT NULL,  
    video VARCHAR(100),  
    PRIMARY KEY (entry_id),  
    FOREIGN KEY(entry_id) REFERENCES entry (id)  
)  
COMMIT
```

Výpis 19: Reprezentace schéma z výpisu 16 v PostgreSQL vygenerovaná rámcem SQLAlchemy

C Výstup automatické denormalizace knihovny pypg

Improvement:

materialized view based on query:

```
SELECT * FROM "product" JOIN "product_variant" ON product_variant.product_id = product.id
      JOIN "variant" ON product_variant.variant_id = variant.id
```

original time spent reading: 292.447328568 ms

original time spent writing: 0 ms

improved time spent reading: 6.41107559204 ms

improved time spent writing: 0 ms

Improvement is recommended. Improvement is 98.0% better than the original.

Improvement:

materialized view based on query:

```
SELECT * FROM "product" JOIN "product_category" ON product_category.product_id =
      product.id JOIN "category" ON product_category.category_id = category.id
```

original time spent reading: 145.055055618 ms

original time spent writing: 0 ms

improved time spent reading: 13.1204128265 ms

improved time spent writing: 0 ms

Improvement is recommended. Improvement is 91.0% better than the original.

Improvement:

materialized view based on query:

```
SELECT * FROM "customer" JOIN "address" ON address.customer_id = customer.id
```

original time spent reading: 229.359149933 ms

original time spent writing: 48.0208396912 ms

improved time spent reading: 13.9319896698 ms

improved time spent writing: 287.025690079 ms

Improvement is not recommended. Improvement is 0% better than the original.

Improvement:

materialized view based on query:

```
SELECT * FROM "product_category" JOIN "category" ON category.id = product_category.
      category_id JOIN "product_variant" ON product_variant.product_id = product_category.
      category_id JOIN "variant" ON variant.id = product_variant.variant_id
```

original time spent reading: 472.248077393 ms

original time spent writing: 0 ms

improved time spent reading: 2012.54701614 ms

improved time spent writing: 0 ms

Improvement is not recommended. Improvement is 0% better than the original.

Improvement:

materialized view based on query:

```
SELECT * FROM "product_variant" JOIN "variant" ON variant.id = product_variant.variant_id
```

original time spent reading: 454.188585281 ms

original time spent writing: 0 ms

improved time spent reading: 531.061649323 ms

improved time spent writing: 0 ms

Improvement is not recommended. Improvement is 0% better than the original.

Výpis 20: Výstup automatické denormalizace po testech popsanych v kapitole 7

D Obsah přiloženého CD

Součástí přílohy na CD jsou zdrojové kódy knihovny pypg, uživatelský manuál, programátorská dokumentace, skript pro ukázkové použití knihovny a skript pro vygenerování ukázkové databáze. Je zde také přiložená elektronická podoba této práce.

txt/ Text diplomové práce v podobě PDF souboru.

src/ Zdrojové kódy knihovny pypg. Skripty `blogapp.py` a `blogapp.sql` pro ukázkové použití knihovny.

man/ Uživatelský manuál knihovny pypg. Doporučuje se přečíst před používáním knihovny.

doc/ Programátorská dokumentace ve formě HTML souborů. Úvodní stránku lze spustit pomocí souboru `index.html` v tomto adresáři.